A Computational Model of Music Composition

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

Citation

Citable link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:17463123

Terms of Use
This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA
A Computational Model of Music Composition

A Dissertation Presented
by
Josiah Wolf Oberholtzer
To
The Department of Music

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in the Subject of
Music Composition

Harvard University
Cambridge, Massachusetts
March 2015
A Computational Model of Music Composition

Abstract

This thesis documents my research into formalized score control, in order to demonstrate a computational model of music composition. When working computationally, models provide an explicit formal description of what objects exist within a given domain, how they behave, and what transformations they afford. The clearer the model becomes, the easier it is to extend and to construct increasingly higher-order abstractions around that model. In other words, a clear computational model of music notation affords the development of a clear model of music composition. The Abjad API for Formalized Score Control, an open-source software library written in the Python programming language and making use of the LilyPond automated typesetting system for graphical output, is presented as such a computational model of music notation. My own compositional modeling work, extending Abjad, is introduced and analyzed in the Python library Consort. A collection of five scores, each implemented as Python packages extending these software libraries, are included. Three of these scores, “Zaira”, “Armilla” and “Ersilia”, rely on Consort as their compositional engine, and are presented along with their complete sources. These scores demonstrate my development as a composer investigating the role of computation in music, and display a variety of large-scale structures and musical textures made possible when working with such modeling tools.
This page intentionally left blank.
# Contents

## I Theory

1 **Introduction**
   1.1 Background & Motivation .................................................. 3
   1.2 Overview of the dissertation ............................................ 5

2 *Abjad: a model of notation* ..................................................... 11
   2.1 Representing objects ........................................................ 14
   2.2 Components ................................................................. 18
      2.2.1 Containers ............................................................. 19
      2.2.2 Parentage .............................................................. 22
      2.2.3 Durations ............................................................... 25
      2.2.4 Named components .................................................. 30
   2.3 Indicators ................................................................. 33
      2.3.1 Indicator scope ....................................................... 35
      2.3.2 Annotations ............................................................ 39
   2.4 Spanners ................................................................. 41
   2.5 Typographic overrides ................................................... 48
   2.6 Selecting components .................................................... 52
      2.6.1 Logical ties ............................................................ 54
      2.6.2 Iteration ............................................................... 56
      2.6.3 Selectors ............................................................... 61
   2.7 Templating & persistence ............................................... 65
   2.8 Other tools ............................................................... 71
      2.8.1 Mutation ............................................................... 71
      2.8.2 Pitch modeling ......................................................... 71
      2.8.3 Parsers ................................................................. 72
      2.8.4 Notation factories .................................................. 73
   2.9 Conclusion ................................................................. 74

3 **Modeling time, rhythm and meter** ........................................ 77
   3.1 Timespans ................................................................. 78
      3.1.1 Annotated timespans ................................................ 80
      3.1.2 Time relations ....................................................... 82
      3.1.3 Operations on timespans ........................................... 85
   3.2 Timespan inventories .................................................... 91
      3.2.1 Operations on timespan inventories ............................... 95
### APPENDIX A  consort source

| A.1 | consort.tools.AbsolutePitchHandler | 437 |
| A.2 | consort.tools.AfterGraceSelectorCallback | 439 |
| A.3 | consort.tools.annotate | 439 |
| A.4 | consort.tools.AttachmentExpression | 441 |
| A.5 | consort.tools.AttachmentHandler | 446 |
| A.6 | consort.tools.AttackPointSignature | 449 |
| A.7 | consort.tools.BoundaryTimespanMaker | 453 |
| A.8 | consort.tools.ChordExpression | 459 |
| A.9 | consort.tools.ChordSpecifier | 465 |
| A.10 | consort.tools.ClefSpanner | 465 |
| A.11 | consort.tools.ClefSpannerExpression | 469 |
| A.12 | consort.tools.ComplexPianoPedalSpanner | 470 |
| A.13 | consort.tools.ComplexTextSpanner | 471 |
| A.14 | consort.tools.CompositeMusicSpecifier | 478 |
| A.15 | consort.tools.CompositeRhythmMaker | 484 |
| A.16 | consort.tools.ConsortTrillSpanner | 490 |
| A.17 | consort.tools.Cursor | 493 |
| A.18 | consort.tools.DependentTimespanMaker | 496 |
| A.19 | consort.tools.DynamicExpression | 502 |
| A.20 | consort.tools.FloodedTimespanMaker | 517 |
| A.21 | consort.tools.GraceHandler | 519 |
| A.22 | consort.tools.HarmonicExpression | 522 |
| A.23 | consort.tools.HashCachingObject | 524 |
| A.24 | consort.tools.KeyClusterExpression | 525 |
| A.25 | consort.tools.LeafExpression | 528 |
| A.26 | consort.tools.LogicalTieExpression | 529 |
| A.27 | consort.tools.MusicSetting | 530 |
| A.28 | consort.tools.MusicSpecifier | 539 |
| A.29 | consort.tools.MusicSpecifierSequence | 545 |
| A.30 | consort.tools.OctavationExpression | 548 |
| A.31 | consort.tools.PerformedTimespan | 549 |
| A.32 | consort.tools.PhrasedSelectorCallback | 553 |
| A.33 | consort.tools.PitchClassPitchHandler | 554 |
| A.34 | consort.tools.PitchHandler | 560 |
| A.35 | consort.tools.PitchOperationSpecifier | 572 |
| A.36 | consort.tools.PitchSpecifier | 579 |
| A.37 | consort.tools.Proportions | 585 |
| A.38 | consort.tools.RatioPartsExpression | 586 |
| A.39 | consort.tools.RegisterInflection | 588 |
| A.40 | consort.tools.RegisterInflectionInventory | 596 |
| A.41 | consort.tools.RegisterSpecifier | 596 |
| A.42 | consort.tools.RegisterSpecifierInventory | 601 |
| A.43 | consort.tools.ScoreTemplate | 602 |
| A.44 | consort.tools.ScoreTemplateManager | 604 |
| A.45 | consort.tools.SeedSession | 609 |
APPENDIX B  

Zaira source code

B.1  Zaira makers source

B.1.1  zaira.makers.Percussion

B.1.2  zaira.makers.ZairaScoreTemplate

B.1.3  zaira.makers.ZairaSegmentMaker

B.2  Zaira materials source

B.2.1  zaira.materials.background_dynamic_attachment_expression

B.2.2  zaira.materials.brazil_nut_music_specifier

B.2.3  zaira.materials.cello_solo_music_specifier

B.2.4  zaira.materials.dense_timespan_maker

B.2.5  zaira.materials.drum_agitation_music_specifier

B.2.6  zaira.materials.drum_brushed_music_specifier

B.2.7  zaira.materials.drum_heartbeat_music_specifier

B.2.8  zaira.materials.drum_used_music_specifier

B.2.9  zaira.materials.drum_on_earth_music_specifier

B.2.10  zaira.materials.erratic_dynamic_attachment_expression

B.2.11  zaira.materials.flourish_rhythm_maker

B.2.12  zaira.materials.granular_timespan_maker

B.2.13  zaira.materials.laissez_vibrer_attachment_expression

B.2.14  zaira.materials.legato_rhythm_maker

B.2.15  zaira.materials.legato_timespan_maker

B.2.16  zaira.materials.metal_agitation_music_specifier

B.2.17  zaira.materials.metal_brushed_music_specifier

B.2.18  zaira.materials.metal_tranquilo_music_specifier

B.2.19  zaira.materials.midground_dynamic_attachment_expression

B.2.20  zaira.materials.legato_timespan_maker

B.2.21  zaira.materials.legato_music_specifier

B.2.22  zaira.materials.legato_time specifier

B.2.23  zaira.materials.legato_time specifier

B.2.24  zaira.materials.legato_time specifier

B.2.25  zaira.materials.legato_time specifier

B.2.26  zaira.materials.legato_time specifier

B.2.27  zaira.materials.legato_time specifier
B.2.28  zaira.materials.percussion_reiteration_musicSpecifier  .................  757
B.2.29  zaira.materials.percussion_superball_musicSpecifier .................  758
B.2.30  zaira.materials.piano_clusters_musicSpecifier .......................  758
B.2.31  zaira.materials.piano_drone_musicSpecifier ..........................  759
B.2.32  zaira.materials.piano_fanfare_musicSpecifier .........................  760
B.2.33  zaira.materials.piano_flourish_musicSpecifier ........................  761
B.2.34  zaira.materials.piano_guero_musicSpecifier ...........................  762
B.2.35  zaira.materials.piano_pedals_musicSpecifier ..........................  763
B.2.36  zaira.materials.piano_prepared_bass_musicSpecifier ..................  763
B.2.37  zaira.materials.piano_prepared_treble_musicSpecifier ...............  764
B.2.38  zaira.materials.proportions ..............................................  766
B.2.39  zaira.materials.registerSpecifier_inventory ...........................  766
B.2.40  zaira.materials.reiterating_rhythm_maker ..................................  769
B.2.41  zaira.materials.sparse_timespan_maker ...................................  769
B.2.42  zaira.materials.string_chord_musicSpecifier ...........................  770
B.2.43  zaira.materials.string_flourish_musicSpecifier .........................  770
B.2.44  zaira.materials.string_trills_musicSpecifier ...........................  771
B.2.45  zaira.materials.string_tutti_overpressure_musicSpecifier .............  772
B.2.46  zaira.materials.string_undergrowth_musicSpecifier .....................  773
B.2.47  zaira.materials.stuttering_rhythm_maker ..................................  774
B.2.48  zaira.materials.sustained_rhythm_maker ..................................  774
B.2.49  zaira.materials.sustained_timespan_maker ................................  774
B.2.50  zaira.materials.tempi ......................................................  775
B.2.51  zaira.materials.time_signatures ............................................  775
B.2.52  zaira.materials.total_duration_in_seconds ................................  776
B.2.53  zaira.materials.tutti_timespan_maker .....................................  776
B.2.54  zaira.materials.undergrowth_rhythm_maker ...............................  776
B.2.55  zaira.materials.wind_airtone_musicSpecifier ............................  777
B.2.56  zaira.materials.wind_keyclick_musicSpecifier ...........................  777
B.2.57  zaira.materials.wind_slap_musicSpecifier .................................  778
B.2.58  zaira.materials.wind_trills_musicSpecifier ................................  779
B.2.59  zaira.materials.wood_agitation_musicSpecifier ..........................  780
B.2.60  zaira.materials.wood_bamboo_musicSpecifier .............................  781

B.3  zaira segments source .........................................................  782
B.3.1  zaira.segments.segment_a ....................................................  782
B.3.2  zaira.segments.segment_b ....................................................  783
B.3.3  zaira.segments.segment_c ....................................................  787
B.3.4  zaira.segments.segment_d ....................................................  793
B.3.5  zaira.segments.segment_e ....................................................  798
B.3.6  zaira.segments.segment_f1 ...................................................  804
B.3.7  zaira.segments.segment_f2 ...................................................  810
B.3.8  zaira.segments.segment_g ....................................................  814
B.3.9  zaira.segments.segment_h ....................................................  818
B.3.10 zaira.segments.segment_i .....................................................  825
B.3.11 zaira.segments.segment_j .....................................................  829
B.3.12 zaira.segments.segment_k .....................................................  832

B.4  zaira stylesheet source ..........................................................  833

x
Appendix C  armilla source code  
C.1 armilla makers source ........................................ 843
C.1.1 armilla.makers.ArmillaScoreTemplate ................. 843
C.1.2 armilla.makers.ArmillaSegmentMaker ............... 846
C.2 armilla materials source ................................ 850
C.2.1 armilla.materials.dense_timespan_maker .......... 850
C.2.2 armilla.materials.intermittent_accents .......... 851
C.2.3 armilla.materials.intermittent_circular .......... 851
C.2.4 armilla.materials.intermittent_glisandi .......... 851
C.2.5 armilla.materials.intermittent_tremoli .......... 852
C.2.6 armilla.materials.intermittent_trills .......... 852
C.2.7 armilla.materials.left_hand_diads_musicSpecifier 853
C.2.8 armilla.materials.left_hand_dietro_musicSpecifier 854
C.2.9 armilla.materials.left_hand_glisandi_musicSpecifier 855
C.2.10 armilla.materials.left_hand_pizzicati_musicSpecifier 856
C.2.11 armilla.materials.left_hand_stasis_musicSpecifier 857
C.2.12 armilla.materials.right_hand_circular_musicSpecifier 857
C.2.13 armilla.materials.right_hand_jete_musicSpecifier 859
C.2.14 armilla.materials.right_hand_overpressure_musicSpecifier 860
C.2.15 armilla.materials.right_hand_pizzicati_musicSpecifier 862
C.2.16 armilla.materials.right_hand_stasis_musicSpecifier 863
C.2.17 armilla.materials.sparse_timespan_maker .......... 864
C.2.18 armilla.materials.sustained_timespan_maker .......... 864
C.2.19 armilla.materials.synchronized_timespan_maker .... 865
C.2.20 armilla.materials.time_signatures ................. 865
C.3 armilla segments source ................................... 866
C.3.1 armilla.segments.segment_a_far_sorr .............. 866
C.3.2 armilla.segments.segment_b_selidor_a ............ 868
C.3.3 armilla.segments.segment_c_wellogy .............. 869
C.3.4 armilla.segments.segment_d_the_long_dune_a .... 871
C.3.5 armilla.segments.segment_e_selidor_b ............ 874
C.3.6 armilla.segments.segment_f_the_isle_of_the_ear .... 875
C.3.7 armilla.segments.segment_g_selidor_c ............ 877
C.3.8 armilla.segments.segment_h_the_long_dune_b .... 878
C.4 armilla stylesheet source ..................................... 881
C.4.1 stylesheet.ily ........................................... 881

Appendix D  ersilia source code  
D.1 ersilia makers source ........................................ 889
D.1.1 ersilia.makers.ErsiliaScoreTemplate ................. 889
D.1.2 ersilia.makers.ErsiliaSegmentMaker ............... 897
D.1.3 ersilia.makers.Percussion ............................. 899
D.2 ersilia materials source .................................. 900
D.2.1 ersilia.materials.abbreviations ................. 900
D.2.2 ersilia.materials.dense_timespan Maker .......... 902
D.2.3 ersilia.materials.guitar_agitato_musicSpecifier .... 903
D.2.4 ersilia.materials.guitar_continuo_musicSpecifier
D.2.5 ersilia.materials.guitar_pointillist_harmonics_musicSpecifier
D.2.6 ersilia.materials.guitar_strummed_musicSpecifier
D.2.7 ersilia.materials.guitar_tremolo_musicSpecifier
D.2.8 ersilia.materials.guitar_undulation_tremolo_musicSpecifier
D.2.9 ersilia.materials.percussion_bamboo_windchimes_musicSpecifier
D.2.10 ersilia.materials.percussion_crotales_flash_musicSpecifier
D.2.11 ersilia.materials.percussion_crotales_interruption_musicSpecifier
D.2.12 ersilia.materials.percussion_low_pedal_musicSpecifier
D.2.13 ersilia.materials.percussion_marimba_agitato_musicSpecifier
D.2.14 ersilia.materials.percussion_marimba_ostinato_musicSpecifier
D.2.15 ersilia.materials.percussion_marimba_tremolo_musicSpecifier
D.2.16 ersilia.materials.percussion_snare_interruption_musicSpecifier
D.2.17 ersilia.materials.percussion_temple_block_fanfare_musicSpecifier
D.2.18 ersilia.materials.percussion_tom_fanfare_musicSpecifier
D.2.19 ersilia.materials.permitted_time_signatures
D.2.20 ersilia.materials.piano_agitato_musicSpecifier
D.2.21 ersilia.materials.piano_arm_cluster_musicSpecifier
D.2.22 ersilia.materials.piano_glissando_musicSpecifier
D.2.23 ersilia.materials.piano_palm_cluster_musicSpecifier
D.2.24 ersilia.materials.piano_pedals_music_setting
D.2.25 ersilia.materials.piano_pointillist_musicSpecifier
D.2.26 ersilia.materials.piano_string_glissando_musicSpecifier
D.2.27 ersilia.materials.piano_tremolo_musicSpecifier
D.2.28 ersilia.materials.pitch_pipe_musicSpecifier
D.2.29 ersilia.materials.saxophone_agitato_musicSpecifier
D.2.30 ersilia.materials.shaker_decelerando_musicSpecifier
D.2.31 ersilia.materials.shaker_sporadic_musicSpecifier
D.2.32 ersilia.materials.shaker_tremolo_musicSpecifier
D.2.33 ersilia.materials.sparse_timespan_maker
D.2.34 ersilia.materials.string_agitato_musicSpecifier
D.2.35 ersilia.materials.string_legato_musicSpecifier
D.2.36 ersilia.materials.string_low_pedal_musicSpecifier
D.2.37 ersilia.materials.string_ostinato_musicSpecifier
D.2.38 ersilia.materials.string_overpressure_musicSpecifier
D.2.39 ersilia.materials.string_pointillist_musicSpecifier
D.2.40 ersilia.materials.string_tremolo_musicSpecifier
D.2.41 ersilia.materials.sustained_timespan_maker
D.2.42 ersilia.materials.tutti_timespan_maker
D.2.43 ersilia.materials.wind_agitato_musicSpecifier
D.2.44 ersilia.materials.wind_contortu_musicSpecifier
D.2.45 ersilia.materials.wind_low_pedal_musicSpecifier
D.2.46 ersilia.materials.wind_ostinato_musicSpecifier
D.2.47 ersilia.materials.wind_pointillist_musicSpecifier
D.2.48 ersilia.materials.wind_tremolo_musicSpecifier
D.3 ersilia.segments_source
D.3.1 ersilia.segments.chemish
D.3.2  ersilia.segments.cut_1 ......................................................... 963
D.3.3  ersilia.segments.cut_2 ......................................................... 964
D.3.4  ersilia.segments.komokome ................................................... 965
D.3.5  ersilia.segments.sort ......................................................... 972
D.4    ersilia stylesheet source .................................................. 978
D.4.1  stylesheet.ily ............................................................... 978

REFERENCES ................................................................. 989
The Astrologer
Hans Holbein
*The Dance of Death*
(c.1527)

For Cindy
A Sorcerer by the power of his magick had subdued all things to himself.
Would he travel? He could fly through space more swiftly than the stars.
Would he eat, drink, and take his pleasure? There was none that did not instantly obey his bidding.
In the whole system of ten million times ten million spheres upon the two and twenty million planes he had his desire.
And with all this he was but himself.
Alas!

- Aleister Crowley, *The Book of Lies*
This page intentionally left blank.
The intent of this dissertation is to document my research into formalized score control\textsuperscript{4,5,18} in order to demonstrate a computational model of music composition. The term formalized score control, inspired by Iannis Xenakis’ seminal text \textit{Formalized Music}\textsuperscript{21,3}, describes the discipline of modeling and manipulating the typography of common practice notation programmatically via software. An emphasis on modeling, here, is crucial. When working computationally, models provide an explicit formal description of what objects exist within a given domain, how they behave, and what transformations they afford. The clearer the model becomes, the easier it is to extend and to construct increasingly higher-order abstractions around that model. That is to say, a clear model of notation – the description of symbols on the page – affords a clear model of composition – how those symbols come to be there.

At the time of this writing, a wide variety of computational models of notation exist\textsuperscript{5,18}, expressing an equally wide range of explicitness or implicitness in their modeling, and providing composers with varying degrees of pro-
grammatic control over those models. For example, many composers make use of vector graphics programs for notation, working in an environment which behaves like a hyper-extended writing desk, but which provides no more semantic understanding of the contents of the page than a physical pencil does. That is to say, the software make no categorical distinction between an oval representing a note head and any other line on the page. Likewise, many composers rely on more traditional notation software which mimics the physical page while providing some model of common practice notation, rigidly enforcing a measure-based understanding of musical score on the artist, while still others make use of the idiosyncratic score-modeling systems often developed hand-in-hand with computer-assisted composition tools. Of course it’s important to understand that I don’t denigrate the use of vector graphics programs to create scores. The relative presence or absence of notation modeling in a tool can be a boon or a hindrance, depending on how it aligns with the needs of the composer using it. While one tool might model notation exquisitely, it still provides programmatic control over only what it can describe, and may even forbid the manipulation of anything beyond that domain. Conversely, another tool, providing only a minimal model of notation, or even none at all, is not so constrained, and while it may not afford programmatic control over the contents of the score at all, it does allow composers to radically reinvent and extend an understanding of Western notation.

My own compositional work now makes extensive use of the *Abjad API for Formalized Score Control*. Abjad, which originated in 2008 as a joint effort between the composers Víctor Adán and Trevor Bača, and to which I have also been contributing significantly for over six years, extends the *Python* programming language with a rich model of common practice notation, and visualizes the scores it models with *LilyPond*, an automated engraving system inspired by the venerable *LaTeX*. On top of Abjad, I have implemented a model of composition in the Python package *Consort*, which allows for the specification of the structure of a score and the processes to be used to create it, and the interpretation of that specification first into Abjad’s notation model, and subsequently into publishable-quality typography via LilyPond. These twin notational and compositional models bring the co-

---

1 Composers often make use of all the above categories of tool in a single score, working programmatically with a computer-assisted composition tool to generate material which is then exported into a more traditional notation editor for page layout and further editing, and finally exported as vector graphics for fine-tuning. This work-flow is not reversible. The changes made to the score as vector graphics cannot be propagated back into the semantic model of the score provided by either the traditional score editor or the computer-assisted composition tool.

2 http://projectabjad.org/

3 http://python.org/

4 http://lilypond.org/
lective knowledge of the computing disciplines to bear on a variety of musical questions: how does one arrange large-scale form? What manner of rhythmic, harmonic and timbral transformations are available? How can one describe formal structural relationships between the objects on a page? How can one quickly audition and vary dense, multi-layered orchestral textures? The chapters that follow investigate these two intertwined forms of modeling, attempting to address some of the musical questions posed here, while paying special attention to those details – often missing from treatises on computer-assisted composition – which straddle the line between high-level and low.

Importantly, this dissertation is not a general survey of the techniques used by composers working in computer-assisted composition, but the work of one composer implementing for the specifics of his own process. This work may well be applicable to others in many ways, but I do not claim universality. It is then, more of a tutorial than anything else, presented in order to explain the architectural decisions, implementations and implications behind a system designed to assist in the composition of music.

1.1 Background & Motivation

Computer-assisted composition has a fairly short history, dating at the time of this writing a little more than half a century. While one could certainly argue about the timeline and origins of the techniques falling under this umbrella – and providing a complete history of computer music is not the goal of this dissertation – it is generally agreed that work began in earnest in the post-war decades of the 20th century.\textsuperscript{15,21} Despite its relative youth as a musical tradition, computer-assisted composition has many practitioners and a tremendous number of tools to work with, both generalized and idiosyncratic.

My personal introduction to computer-assisted composition began with Iannis Xenakis’ mid-century massed string works,\textsuperscript{21} and, combined with a life-long interest in electro-acoustic music dating back to my serendipitous childhood exposure to a copy of Ivan Tcherepnin’s \textit{Flores Musicales / Five Songs / Santur Live!}, quickly progressed through study of Curtis Roads’ granular explorations,\textsuperscript{16} Trevor Wishart’s schema for textural transformations,\textsuperscript{20} and Kaaija Saariaho’s analytic orchestrations. While I’ve turned to other music and other considerations since reading these texts and listening to these works, they instilled in me a confidence that somehow computation could be useful for creating the sort of music I wanted to hear but was consistently unable to write, simply sitting at my desk.\textsuperscript{5}

\textsuperscript{5}The jury is still out, but I persist anyway.
Figure 1.1: The first page of mbrsri (2006), an unfinished tablature score for one to twelve string performers. This represents an early attempt of mine at computationally modeling massed performers with timespans to create an evolving "granular" texture. The instructions for the score were created through a variety of Max/MSP patches which painstakingly converted noise functions into text files which I then collated into spreadsheets. The score itself was drawn by hand on size A1 graph paper with Rapidographs. I completed seventeen of the intended fifty-three pages before stopping for the sake of my wrists and due a general lack of faith in the project. While still incomplete, the concerns that motivated this score remain with me.
My earliest experiments with computer-assisted composition involved Max/MSP, writing patches to generate tabular data from various noise functions which I collected into spreadsheets and painstakingly notated by hand. That work culminated at the end of my undergraduate study in 2006 with the never-completed score *mbrsi* – excerpted in figure 1.1, with one spreadsheet “source” in figure 1.2 –, whose textural fabric of massed strings attempted to crystallize the possibilities I felt in granular synthesis at the time: a kind of sparkling, nervous energy hovering between a flame and the strange Bezier-cloud outlined by starling swarms. Unfortunately, the process of simply notating the piece by hand was gruelling, and effectively forbade any effort at compositional exploration, let alone revising. In short, while this early work did take lengths to model a compositional process, it lacked any notational model beyond my own hands. And in the absence of any automated typesetting tool, the project collapsed under its inherent labor difficulties. I set *mbrsi* aside and spent the next few years looking for alternative approaches to both formalizing large-scale structure in score, and automating the typesetting of those structures into notation, going so far as to attempt scripting various vector graphics programs directly in order to establish a formalized model of music notation which I could use to complete the project.

A number of years later, at the beginning of my graduate studies, I was introduced to first LilyPond, and then Abjad. LilyPond, much like LaTeX, is a command-line tool which takes a text file comprising various commands which describe the content of a musical document, and converts those commands into graphic representations of music notation, generally as a PDF. This working modality is often referred to as “what you see is what you mean”: the user of the system edits with the intent to describe their desired structure, rather than manipulating a representation of the end-product itself. The discovery of such a plain-text-based way of working was a revelation to me, precisely because plain text is eminently susceptible to manipulation. One would be hard-pressed to find any programming languages which cannot generate plain-text files. While hardly easy – and I simply couldn’t have known at the time how far off that impression was – a way forward finally seemed possible: I could develop a system to model notation, using LilyPond as its typesetting engine. Luckily, Víctor Adán and Trevor Baća had reached an identical conclusion sometime before, and had already begun the work of implementing a notation model in the Python programming language, using LilyPond as the primary output. And now, nearly six years later, this work continues, unabated.

Of course there were already a tremendous number of composition tools contemporary with my entry into the field, such as OpenMusic,\(^2\) PWGL,\(^{10, 9}\) Common Lisp Music,\(^{17}\) and somewhat later the BACH\(^1\) library for use with modern versions of Max, as well as a whole host of more marginal experiments including LilyCollider, which
**Figure 1.2:** Page one of fifty-three from a spreadsheet containing fingering instructions for six of the twelve string performers in *mōryō* (2006). Each box consists of the instructions for one performer, over the course of one measure. The instructions in the upper-left box here correspond to the upper-most row of boxes in figure 1.1.
provides an interface between SuperCollider \cite{supercollider} – generally used for live electronics – and LilyPond. Why develop yet another? The answer is both pragmatic, and also simply preferential. I wanted a tool usable from the command-line, something transparently interoperative with the Unix ecosystem, not a graphical application. I wanted it to be open-source, so I could inspect it, break it, and come to understand all of its mechanisms. I wanted it to be free, so I could use it anywhere. And I wanted it to be written in an imperative language, so I could learn it more quickly. Abjad fit all of these desires.

1.2 Overview of the dissertation

This dissertation consists of six chapters of prose – including this chapter –, followed by five chapters each presenting a score, and four extensive appendices comprising complete code listings for the implementations of the three most recent scores and of Consort. Please consult the source for Abjad version 2.16\footnote{https://github.com/Abjad/abjad/releases/tag/2.16} directly, as it is far too large to include here.

The first six chapters discuss formalized score control, gradually laying out and building upon the concepts necessary to understand the process of modeling notation and composition computationally. Chapter 2 presents an overview of Abjad, detailing its structure and usage in the creation of scores. The objects comprising Abjad’s core notation model, \textit{components, indicators} and \textit{spanners}, are introduced, and the techniques by which those objects are aggregated into scores, inspected, selected, iterated over, mutated, persisted to disk, and visualized are demonstrated at length. Chapter 3 expands on chapter 2, discussing various models of musical time implemented in Abjad, and introducing many of the tools and techniques employed in Consort to create large-scale musical works. Timespans – objects which model a duration of time positioned along a timeline without regard for score hierarchy –, as well as massed collections of timespans, and operations applied against both single timespans and those collections, are presented as a means for modeling high-level phrasing and score structure. Highly-configurable factory classes for programmatically creating both timespan and rhythmic structures are introduced along with a hierarchical model of meter which affords coordination between the two models of time. Chapter 4 analyzes the mechanisms implemented in Consort’s model of composition to specify the structure of musical score at a high level and to interpret those specifications into completely-notated segments of music. The analysis proceeds step-by-step through the process of specification, introducing the reader to Consort’s segment-makers, music specifiers and music settings. The discussion then turns to interpretation, divided into rhythmic- and non-rhythmic stages. The mechanisms by
which the music settings comprising a score specification are gradually resolved into a maquette consisting of annotated timespans, that maquette interpreted into a score, and grace notes, pitches and attachments applied against that interpreted score are presented, providing a link back to the iteration and selection techniques presented in chapter 2 and the timespan transformations demonstrated in chapter 3. Chapter 5 proposes standardized solutions to practical concerns surrounding the composition of scores in software, such as score package layout on the file-system, typesetting workflows, version control, and testing. The means by which an individual score segment, as described in chapter 4, is combined with many other segments to create a complete work is presented, and the document preparation techniques necessary for automating the extraction of parts in LilyPond and producing the various non-musical component documents of a score with \LaTeX are also introduced. Chapter 6, the conclusion to the prose portion of the dissertation, summarizes the previously presented research, suggests some implications and areas for improvement, and proposes directions for future work.

The remaining chapters consist of five scores, all composed computationally with Abjad and LilyPond, both prior to and after the development of Consort. Chapter 7 presents *Aurora* (2011), for string orchestra, which represents the results of my first formal research into composition with timespan structures, as well as my first formalized work which allowed multiple distinct textures to overlap. This score implemented a very rudimentary version of the “specify & interpret” pattern enshrined in Consort. Chapter 8 presents *Plague Water* (2014), for baritone saxophone, electric guitar, piano and percussion, my first attempt at composing scores in segments through explicit specification and interpretation, and whose code-base provided Consort’s original foundation. Chapters 9, 10 and 11 present a set of pieces, *Invisible Cities (i): Zaira* (2014), for eight players, *Invisible Cities (ii): Armilla* (2015), for viola duet, and *Invisible Cities (iii): Ersilia* (2015), for chamber orchestra, all implemented via Consort. These three scores demonstrate Consort’s flexibility as a tool for structuring both large and small ensemble works of widely divergent textures, notated both conservatively – as in *Zaira* and *Ersilia* – and more unconventionally, with tablature as in *Armilla*.

Finally, the appendices contain the source to all classes and functions implemented in Consort – as of the time of this writing –, as well as the source to all material and segment definitions, as well as any LilyPond stylesheets, for the three Invisible Cities scores. My sincere hope is that these appendices provide a concrete, “reverse-engineerable” even, link between the descriptions of my research into formalized score control and the scores written with these tools.
Abjad: a model of notation

Abjad is an open-source Python package extending the Python programming language with a computational model of music notation. Abjad allows its users to construct scores in an object-oriented fashion and to visualize their work as publication-quality notation at any point during the composition process. Importantly, Abjad is not a stand-alone application and has no GUI – no graphic user interface – unlike many other notation- or composition-modeling projects such as PWGL, OpenMusic, BACH, and so forth. Instead, all work with Abjad is done in the command-line, either directly in an interactive Python console session, or by writing modules of code to be imported into one another and executed by Python’s interpreter. Nearly all of the code examples in the body of this document are presented as part of a single interactive Python console session simply because Python’s interactive console clearly distinguishes input from output. Lines preceded by » will be passed to Python for interpretation. Those preceded by . . . indicate the continuation of a
We can import Abjad into Python with the following incantation:

```python
>>> import abjad
```

Abjad is implemented as a collection of over 900 public classes and functions spread across nearly 40 subpackages, totally almost 200,000 lines of source code. Many of these classes and functions are available immediately within Abjad’s root namespace.\(^1\)

We can enumerate these “top-level” names by calling Python’s built-in `dir()` function on the imported `abjad` module object. After stripping out all “private”\(^2\) names – those starting with underscores – Abjad’s top-level names can be printed to Python’s interactive console:

```python
>>> abjad_names = dir(abjad)
>>> abjad_names = [x for x in abjad_names if not x.startswith('_')]
>>> print(abjad_names)
```

Names beginning with uppercase letters represent classes. These include many of the musical objects composers would likely create by hand. Most of these classes have direct notational analogs on the page – e.g. `Note`, `Rest` and `Chord` –, or map to common structural or typographic concepts in Abjad’s primary typesetting engine, LilyPond, such as `Voice`, `Markup` and `Skip`. Lowercase names ending in the string “tools” represent additional libraries or

---

\(^1\)In programming, **namespace** represents contexts in which a particular collection of **names**, or identifiers, are grouped together, with each name referencing some code object, such as a class, function or number. Python makes extensive use of namespaces, treating every module – any file containing code – as a namespace, as well as any class, the bodies of functions and so forth. Python’s namespaces are implemented as dictionaries, a type of data structure which maps a unique set of keys to values. Each name in a Python namespace therefore appears only, although multiple names may be bound to the same value.

\(^2\)Unlike many other languages, such as C++ or Java, Python does not enforce the concept of “public” versus “private” objects at the language level. Any object in any namespace can be accessed by any actor at any time. Instead, Python programmers use naming conventions to indicate which objects represent components of public interfaces, and which objects should be considered private, although still accessible, implementation details.
subpackages within Abjad. These subpackages group together related functionality within a common namespace. For example, Abjad’s `pitchtools` subpackage contains dozens of classes and functions solely for modeling pitch objects, collections of those objects, and their transformations:

```python
>>> pitchtools_names = dir(abjad.pitchtools)
>>> pitchtools_names = [x for x in pitchtools_names if not x.startswith('_')]
>>> print(pitchtools_names)
['Accidental', 'Interval', 'IntervalClass', 'IntervalClassSegment', 'IntervalClassSet', 'IntervalClassVector', 'IntervalSegment', 'IntervalSet', 'IntervalVector', 'Inversion', 'Multiplication', 'NamedInterval', 'NamedIntervalClass', 'NamedInversionEquivalentIntervalClass', 'NamedPitch', 'NamedPitchClass', 'NumberedInterval', 'NumberedIntervalClass', 'NumberedPitch', 'NumberedPitchClass', 'NumberedPitchClassColorMap', 'Octave', 'Pitch', 'PitchArray', 'PitchArrayCell', 'PitchArrayColumn', 'PitchArrayInventory', 'PitchArrayRow', 'PitchClass', 'PitchClassSegment', 'PitchClassSet', 'PitchClassTree', 'PitchClassVector', 'PitchOperation', 'PitchRange', 'PitchRangeInventory', 'PitchSegment', 'PitchSet', 'PitchVector', 'Registration', 'RegistrationComponent', 'RegistrationInventory', 'Retrogression', 'Rotation', 'Segment', 'Set', 'StaffPosition', 'Transposition', 'TwelveToneRow', 'Vector', 'instantiate_pitch_and_interval_test_collection', 'inventory_inversion_equivalent_named_interval_classes', 'list_named_pitches_in_expr', 'list_numbered_interval_numbers_pairwise', 'list_numbered_inversion_equivalent_interval_classes_pairwise', 'list_ordered_named_pitch_pairs_from_expr_1_to_expr_2', 'list_pitch_numbers_in_expr', 'list_unordered_named_pitch_pairs_in_expr', 'sort_named_pitch_carriers_in_expr', 'transpose_named_pitch_by_numbered_interval_and_respell', 'transpose_pitch_carrier_by_interval', 'transpose_pitch_class_number_to_pitch_number_neighbor', 'transpose_pitch_expr_into_pitch_range', 'transpose_pitch_number_by_octave_transposition_mapping', 'yield_all_pitch_class_sets']
```

Many of these subpackages implement opinionated collections of compositional devices, certainly useful for some composers but by no means considered core to Abjad’s notational model. For example, Abjad’s `quantizationtools` provides a rhythmic quantizer inspired by Paul Nauert’s concept of “Q-grids”\(^\text{13}\), `rhythmtree tools` parses a Lisp-like “RTM-syntax” – which should be familiar to anyone working with OpenMusic\(^2\), PWGL\(^9\,9\) or Bach\(^1\), – into object-oriented rhythm-tree data structures, and the `sieves` subpackage models Iannis Xenakis’ pitch sieves\(^21\). Still other subpackages implement more “objective” functionality, such as `mathtools` or `durationtools`. Many of the remaining subpackages collect together related musical and typographic classes and functions, e.g. `scoretools`, `spannertools`, `markuptools`, and `indicatortools`. Others simply exist for “internal” use by Abjad’s developers, assisting in the development and maintenance of the system, such as `abctools`, `developerscripttools`, `documentationtools` and `systemtools`.

The remaining names in Abjad’s namespace represent its “top-level” functions, including `attach()`, `detach()`, `inspect_()`, `iterate()`, `mutate()`, `new()`, `override()`, `parse()`, `persist()`, `play()`, `select()`, and – most impor-
tantly – `show()`. These functions provide a powerful set of tools for interacting with Abjad’s notational objects and factories, and will be explained in detail as we encounter each of them.

Finally, simply for the sake of brevity, this chapter – and each subsequent chapter – will behave as though the contents of Abjad’s namespace has been imported into Python’s global namespace:

```python
from abjad import *
```

## 2.1 Representing objects

In Python, everything is an object, even integers, boolean values and functions. Because Python is also interpreted and can be used interactively, those objects must be representable in an interpreter session, which is necessarily textual. An object’s textual, or `string`, representation can take a variety of forms, and that same object may even specify explicitly how it should be represented textually, just as it might specify how it should behave in relation to such operations as addition, multiplication, or iteration.

Python classes often specify their behavior in terms of `protocols`, sets of methods which explain the kinds of behaviors a class is able to engage in. This explanation takes place when a class implements or overrides “dunder” methods – double-underscore methods – such as `__repr__()` which provides an object’s `interpreter representation` in cooperation with Python’s built-in `repr()` function, and `__str__()` which provides an object’s string representation in cooperation with Python’s built-in `str()` function. By overriding these dunder methods, a class may specify its own version of string or interpreter representation behavior. Like public and private names, Python protocols are also established by convention rather than enforced at the language level, in contrast to similar mechanisms in other languages such as Java’s interfaces.

For example, a simple `Foo` class, which overrides neither `__str__()` nor `__repr__()` may be defined and instantiated:

```python
>>> class Foo(object):
...     pass
...
>>> foo = Foo()
```

When printed to the terminal, cast as a string, and represented, Python’s default object representation is used, giving the module and name of the class as well as its memory location. Note that when calling Python’s `print()` function on an object, that object is converted to a string and then printed to the console, resulting in text without quotation marks:
Defining a new class, `Foo2`, which overrides both string and interpreter representation provides customized output. Note that printing the `Foo2` instance relies on its string representation, rather than its interpreter representation, while simply referencing the instantiated `Foo2` object uses its interpreter representation – its “repr”:

```python
>>> class Foo2(object):
...     def __repr__(self):
...         return '<I am Foo2>'
...     def __str__(self):
...         return 'foofoo'

>>> foo2 = Foo2()
>>> foo2
<I am Foo2>
>>> print(foo2)
foofoo
>>> repr(foo2)
'<I am Foo2>'
>>> str(foo2)
'foofoo'
```

Defining a third version of `Foo` with an overridden `__format__()` method allows for creating alternate string representations. By calling Python’s built-in `format()` function on a `Foo3` instance, either the normal string representation or a “special” representation can be created. This formatting behavior can also be extended to support an arbitrary number of format specifications:

```python
>>> class Foo3(object):
...     def __format__(self, format_specification=''):  # Note the default argument
...         if format_specification == 'special':
...             return '~~~ special foo format ~~~~'
...         return str(self)
...     def __repr__(self):
...         return '<I am Foo3>'
...     def __str__(self):
...         return 'foofoofoo'

>>> foo3 = Foo3()
>>> foo3
<I am Foo3>
```
Many Abjad objects have multiple possible string representations. For example, a `Note` object can be represented by its normal interpreter representation, as a LilyPond string, or by its `storage format` – a more verbose, potentially multi-line string representation generally used when persisting complex objects to disk.

For some Abjad classes – especially those representing glyphs in music notation – their default string format is their LilyPond string representation. Other classes – especially Abjad’s various highly-configurable notation factory classes – default their string format to their `storage format`, allowing for complex but still human-readable interpreter output. The idiom `print(format(...))` is used extensively throughout this document – especially in later chapters – to quickly display these multi-line storage formats:
Finally, in the spirit of Python’s protocols and dunder conventions, Abjad provides its own *illustration protocol* and corresponding *__illustrate__()* dunder method. This method works with Abjad’s top-level *show()* function to generate graphic realizations of illustrable Python objects by automatically formatting those objects into LilyPond syntax and passing the resulting input to LilyPond for typesetting. Abjad’s *show()* function will be used throughout this document to create and display music notation and other graphic representations of the objects discussed here:
2.2 Components

Abjad models notated musical scores as a tree consisting of components, indicators and spanners. All objects appearing in a score fall into one of those three categories. The most important and most complex of the three groups, components, make up the nodes of the score tree, treating the score as an acyclic directed rooted graph, or arborescence, rooted on a single component. All components inherit from Abjad’s Component class, which encapsulates logic crucial to maintaining the correctness of the score tree, and affords behaviors common to all component subclasses including illustration, formatting and naming. Abjad divides score components into containers – those components which may contain other other components, such as staves, voices, tuplets, measures and so forth – and leaves – those components which may contain no other components, such as notes, chords and rests, as well as LilyPond’s multi-measure rests and non-printing typographic skips. Additionally, components can each be instantiated from a variety of input, from parsable LilyPond syntax strings to parametric arguments – both numeric and explicitly object-modeled via pitch and duration objects. For example, we can instantiate and illustrate a middle-C quarter note:

```python
>>> note = Note("c'4")
>>> show(note)
```

![Middle-C Quarter Note]

A half-note rest:

```python
>>> rest = Rest((1, 2))
>>> show(rest)
```

![Half-Note Rest]

And a D-minor chord:

```python
>>> chord = Chord(  
... [NamedPitch("d'"), NamedPitch("f'"), NamedPitch("a'")],  
... Duration(1, 4),  
... )
>>> show(chord)
```

![D-Minor Chord]
Each of the above three leaf instantiations demonstrates a different means of creating score objects, none of which is unique to the particular class they have been demonstrated with here. The Note instantiation demonstrates creating a score object via LilyPond syntax, the rest instantiation demonstrates creating a score object from sequences of integers – in this case the pair (1, 2), representing the duration of a half-note – and the chord instantiation demonstrates using explicit NamedPitch and Duration objects. The latter two instantiation techniques afford parametric approaches, while the LilyPond syntax allows for a great degree of expressivity as score components can be created with articulations, dynamics and other indicators already attached:

```python
>>> fancy_chord = Chord(r"g' bf'? d''>4. -\accent -\staccato \markup { \italic gently }")
>>> show(fancy_chord)

```

2.2.1 Containers

Abjad’s container classes – measures, tuplets, voice, staves, staff groups, scores and so forth – implement Python’s mutable sequence protocol, allowing them to be used transparently as though they were lists. Components can be appended, extended, or inserted into containers. Containers can also be instantiated with lists of components to be inserted as one of their instantiation arguments. Note in the previous notation examples that the LilyPond output always shows a $4/4$ time signature even though none of the created leaves takes up a whole note’s duration. LilyPond simply treats all music as $4/4$ unless explicitly instructed otherwise. We can make the $4/4$ time signature explicit by instantiating a $4/4$ measure object to contain the three previously-created leaf instances, with the three leaves passed in a list as an instantiation argument to the new measure:

```python
>>> measure = Measure((4, 4), [note, rest, chord])
>>> show(measure)

```

The measure can also be represented as a graph, clarifying the containment:

```python
>>> graph(measure)
```
Like lists, containers can be indexed into, measured for length and iterated over. For example, the second component contained by the measure can be selected by subscripting the measure with the index 1:

```python
>>> measure[1]
Rest('r2')
```

That second component – the half-note rest – can be replaced by another container, a triplet:

```python
>>> outer_tuplet = Tuplet((2, 3), "a'4 b4 cs'4")
>>> measure[1] = outer_tuplet
>>> show(measure)
```

```plaintext
\[\begin{array}{c}
\text{Measure} \\
\text{4/4} \\
\text{Tuplet} \\
\text{* 2/3} \\
\text{Note} \\
\text{c'4} \\
\text{Note} \\
\text{a'4} \\
\text{Note} \\
\text{b4} \\
\text{Note} \\
\text{cs'4} \\
\text{Chord} \\
\text{<d' f' a'>4}
\end{array}\]
```

```plaintext
>>> graph(measure)
```

Furthermore, the last two leaves in the new triplet can be replaced by a quintuplet using Python's `slice assignment` syntax. In the following code, the text `outer_tuplet[1:]` references the selection of components in the original triplet starting from the second component and going “to the end” – effectively the second and third item, as there are only three. That selection is then replaced by a list containing a quintuplet, substituting the contents of one sequence of components for the contents of another.

```python
>>> inner_tuplet = Tuplet((4, 5), "b'8 a'8. g'16 f'32 e'8..")
>>> outer_tuplet[1:] = [inner_tuplet]
>>> show(measure)
```

```plaintext
\[\begin{array}{c}
\text{Measure} \\
\text{4/4} \\
\text{Tuplet} \\
\text{* 2/3} \\
\text{Note} \\
\text{c'4} \\
\text{Note} \\
\text{a'4} \\
\text{Note} \\
\text{b4} \\
\text{Note} \\
\text{cs'4} \\
\text{Chord} \\
\text{<d' f' a'>4} \\
\text{Note} \\
\text{c'4} \\
\text{Note} \\
\text{a'4}
\end{array}\]
```

3. Like most programming languages, Python counts from zero. Therefore the “first” item in an ordered collection is indexed by the number zero, not one.
The measure can be inspected for its length by using Python’s built-in `len()` function. Note that this returns the number of components *immediately* contained by the measure – 3 –, but not the total number of components, total number of leaves or total duration. The latter queries can be satisfied by other means.

```
>>> len(measure)
3
```

Iterating over the contents of the 4/4 measure yields only the top-most components in that container – its immediate children:

```
>>> for component in measure:
...     component
...
    Note("c'4")
    Tuplet(Multiplier(2, 3), "a'4 { 4/5 b'8 a'8. g'16 f'32 e'8.. }")
    Chord("<d' f' a'4>")
```

To select all of the leaves of a container, recursion must be used, as any container may contain other containers, and those in turn still more containers. To mitigate this complexity, every container provides a `select_leaves()` method, which returns a selection of the bottom-most leaf instances in the subtree rooted at that container:

```
>>> for leaf in measure.select_leaves():
...     leaf
...
    Note("c'4")
    Note("a'4")
    Note("b'8")
    Note("a'8.")
    Note("g'16")
    Note("f'32")
    Note("e'8..")
    Chord("<d' f' a'4>")
```
Note that Abjad’s Chord class aggregates multiple note heads rather than notes. Likewise Abjad’s Note class aggregates a single note head. While chords implement containment in terms of note heads – delegating to a dedicated note head inventory – they are not themselves “containers” in the same sense that a voice, measure or tuplet are containers. It is important here to differentiate between the concept of a “chord” as a single duration paired with a collection of pitches and a “chord” as the sounding sonority at some given point in a piece spread over some number of voices. Abjad always makes use of the former rather than the latter.

```python
>>> note_head_inventory = chord.note_heads
>>> for note_head in note_head_inventory:
...     note_head
...
NoteHead("d'")
NoteHead("f'")
NoteHead("a'")

>>> chord.note_heads.append(NamedPitch("c''"))
>>> show(chord)
```

2.2.2 Parentage

The container that contains a given component is called its parent, and the components contained in a container are known as that container’s children. Every component may have one and only one parent container, although a component may also have no parent – a null parent reference. Any component whose parent is null is necessarily the root of its own score tree. Furthermore, no component may appear in its own proper parentage – the sequence of components comprising the unique path from a given component to the root of its tree, excepting the component itself – as this would induce reference cycles within the score tree.

Abjad object-models the concept of parentage explicitly as a Parentage selection class, accessible via the component inspector which exposes a given component’s inspection interface: a collection of methods for accessing information about that component, many of which depend on that component’s position within the score hierarchy, including the component’s parentage or duration. An inspector can be instantiated by calling Abjad’s top-level inspect() function on a component.\footnote{Abjad uses inspect\_() rather than inspect() as Python already provides an inspect module for object introspection, and it is considered bad practice to overwrite names of functions, classes or modules found in the standard library such as set or object. When a name conflict would occur, projects traditionally append an underscore to the name they wish to use.} Consider the parentage for the first eighth note of the inner quintuplet in the previously
created measure. This eighth-note’s parentage – *improper* by default – includes itself, its immediate quintuplet parent, its triplet grandparent and the \( \frac{3}{4} \) measure as its great-grandparent.⁵

```python
>>> inner_b_eighth = measure[1][1][0]
>>> inspector = inspect_(inner_b_eighth)
>>> parentage = inspector.get_parentage()
>>> parentage.parent
Tuplet(Multiplier(4, 5), "b'8 a'8. g'16 f'32 e'8..")
```

```python
>>> for component in parentage:
...     component
...     ...
Note("b'8")
Tuplet(Multiplier(4, 5), "b'8 a'8. g'16 f'32 e'8..")
Tuplet(Multiplier(2, 3), "a'4 { 4/5 b'8 a'8. g'16 f'32 e'8.. }")
Measure((4, 4), "c'4 { 2/3 a'4 { 4/5 b'8 a'8. g'16 f'32 e'8.. } } <d' f' a' c''>4")
```

By default, every component’s parent is null, represented in Python by the `None` object:

```python
>>> whole_note = Note("c'1")
>>> inspect_(whole_note).get_parentage().parent is None
True
```

Inserting the above whole note into a container sets the whole note’s parent to that container. Additionally, the container is now aware that it contains the whole note, demonstrating the bi-directional references inherent to Abjad’s score tree model:

```python
>>> container_one = Container()
>>> container_one.append(whole_note)
>>> inspect_(whole_note).get_parentage().parent is container_one
True
```

```python
>>> whole_note in container_one
True
```

```python
>>> print(format(container_one))
{  
c'1
}
```

Inserting the same whole note into a different container removes it from the first, demonstrating that components can only exist in one container at a time:

⁵ The component summary strings in the interpreter representations of the components in the parentage are not valid LilyPond syntax, but simply a concise LilyPond-like syntax.
>>> container_two = Container()
>>> container_two.append(whole_note)
>>> inspect_(whole_note).get_parentage().parent is container_two
True

>>> whole_note in container_two
True

>>> whole_note in container_one
False

>>> print(format(container_two))
{
  c'1
}

>>> print(format(container_one))
{
}

Likewise, while the first container can be inserted into the second, the second container cannot be inserted back
into the first. Such an arrangement would cause the second container to become its own grandparent, and therefore
generates an error:

>>> container_two.append(container_one)
>>> container_one.append(container_two)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "~/Users/josiah/Development/abjad/abjad/tools/scoretools/Container.py", line 1062, in append
    self.__setitem__(slice(len(self), len(self)), [component])
  File "~/Users/josiah/Development/abjad/abjad/tools/scoretools/Container.py", line 218, in __setitem__
    return self._set_item(i, expr)
  File "~/Users/josiah/Development/abjad/abjad/tools/scoretools/Container.py", line 561, in _set_item
    raise ParentageError('Attempted to induce cycles.')
ParentageError: Attempted to induce cycles.

Why should we be concerned with reference cycles? When we create score by hand, notes – no matter how we may
have arrived at them or conceived of them creatively – are ultimately simply black marks on a page. However, when
working with computers, we can choose how to model musical objects. Again, a “note” could simply exist as an
amalgam of dots and lines in a vector-graphics program, in which case it may not be possible to know where or
even what that note or any other note is except in terms of dots and lines – the affordances provided by that vector-
graphics editor. In that case, there is no semantic model of music, only a typographic model devoid of explicit
musical relationships. Alternatively, we can model explicitly, in which case notes exist as "objects" in a rich network
of references. In this latter case, certain conditions must be maintained to protect the integrity of the reference
network. Depending on the implementation, a note likely cannot be in two places at once. That is the certainly the case in Abjad’s model of musical notation: notes can only have one parent, and cannot exist in more than one parent. Note that these restrictions mainly derive from the fact that notes, like other scores components, are both mutable – that is, changeable after they have been instantiated, and therefore stateful – and possess many references to objects “outside” of themselves. In contrast, Abjad’s pitch and duration classes have no state – they are immutable – and reference no other objects. They can therefore appear in many places within the same score. Arbitrarily many score components can reference the same duration in memory, much as arbitrarily many objects can reference the same integer. Such an arrangement is often referred to as a flyweight.

2.2.3 Durations

Every Abjad score component may be expressed in terms of its duration, as well as its start offset relative to the score origin and, by extension, its timespan – the span of time bounded by its start offset and stop offset. Abjad models all such duration objects as rationals: ratios of real numbers, e.g. \( \frac{1}{2}, \frac{7}{23} \text{ or } \frac{16}{1} \). This concern arises from the realization that all durations and offsets expressible in Western common practice notation are rational, rather than floating-point or any other representation. Necessarily, Abjad’s Duration, Offset and Multiplier classes all derive from Python’s fractions.Fraction class. Both Offset and Multiplier are little more than aliases to Abjad’s Duration class. However, their use throughout Abjad’s code-base, and those projects heavily dependent upon Abjad, such as Consort, greatly clarify compositional intent and increase the source’s legibility.

Abjad’s Duration class extends Python’s Fraction class with a number of new initialization patterns and a variety of notation-specific properties. For example, a \( \frac{7}{16} \) duration can be instantiated from a numerator/denominator pair, or from a LilyPond syntax duration string:

```
>>> Duration(7, 16)
Duration(7, 16)

>>> Duration.from_lilypond_duration_string('4..')
Duration(7, 16)
```

Duration objects can provide information about how they should be interpreted notationally, including how many dots or flags they would have if used to instantiate a leaf. A \( \frac{7}{16} \) duration, without resorting to tuplets or ties, might be represented in notation by a double-dotted quarter note, thus giving a dot count of 2 and a flag count of 0:

```
>>> Duration(7, 16).dot_count
2
```
As mentioned in subsection 2.2.2, duration are immutable. Like other immutable objects in Python – e.g. integers and string –, they raise errors when anything attempts to alter them in anyway:

```python
>>> string = 'abcdefghi'
>>> string[1:-1] = 'xyz'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

```python
>>> Duration(3, 4).numerator = 7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

```python
>>> Duration(1, 2).nonexistant_property = 'This will never work'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Duration' object has no attribute 'nonexistant_property'
```

Additionally, some mathematical operations between offsets and durations provide results typed in a musically-sensible fashion. For example, the difference between two offsets returns a duration, while a duration added to an offset returns another offset:

```python
>>> offset_one = Offset(1, 2)
>>> offset_two = Offset(7, 4)
>>> offset_two - offset_one
Duration(5, 4)
```

```python
>>> offset_two + Duration(3, 8)
Offset(17, 8)
```

The duration of each leaf derives from the product of its written duration – a duration representing the actual glyphs used in the score as represented by some combination of note heads, stems, beams and dots – and their prolation – the cumulative product of all of the duration multipliers of the containers in a component’s proper parentage.

Consider again the measure created earlier in subsection 2.2.1:
The B eighth-note starting the quintuplet is nested within two different tuplets with different multipliers. While the eighth-note’s parentage object can certainly calculate its prolation automatically, we will calculate the prolation here “by hand” in order to demonstrate the technique:

```python
>>> parentage = inspect_.__(inner_b_eighth).get_parentage()
>>> print(parentage.prolation)
Multiplier(8, 15)
```

```python
>>> by_hand_prolation = 1
>>> for parent in parentage[:]:
...     if isinstance(parent, Tuplet):
...         by_hand_prolation = by_hand_prolation * parent.multiplier
... >>> print(by_hand_prolation)
Multiplier(8, 15)
```

By multiplying each leaf’s written duration by its prolation, we can determine that leaf’s actual, or prolated, duration.

Note too that this observation conforms to the results of the inspector’s `get_duration()` method:

```python
>>> for leaf in measure.select_leaves():
...     inspector = inspect__(leaf)
...     written_duration = leaf.written_duration
...     prolation = inspector.get_parentage().prolation
...     actual_duration = inspector.get_duration()
...     string = '{!r}: {!s} * {!s} = {!s}'
...     string = string.format(leaf, written_duration, prolation, actual_duration)
...     print(string)
```

```
Note("c'4"): 1/4 * 1 = 1/4
Note("a'4"): 1/4 * 2/3 = 1/6
Note("b'8"): 1/8 * 8/15 = 1/15
Note("a'8. "): 3/16 * 8/15 = 1/10
Note("g'16"): 1/16 * 8/15 = 1/30
Note("f'32"): 1/32 * 8/15 = 1/60
Note("e'8.. "): 7/32 * 8/15 = 7/60
Chord("<d' f' a' c''>4"): 1/4 * 1 = 1/4
```

Written durations must be assignable. Assignability describes the set of durations describable in Western common practice notation solely through combining a single note head, its flags and dots, without recourse to ties or tuplets. Any rational duration \(\frac{n}{d}\) is considered assignable when and only when it adheres to the form

\[
\frac{2^k \cdot (2^u - f)}{2^v}
\]  

(2.1)
where \( u, v \) and \( k \) are nonnegative integers, \( j \leq u \) and \( j \) is either 1 or 0. Assignability guarantees that a duration’s denominator is always a positive power-of-two integer, such as 1, 2, 4, 8, 16 and so forth, and therefore precludes durations such as \( \frac{1}{3} \) or \( \frac{2}{5} \). Likewise, assignability permits numerators such as 1, 2, 3, 4, 6, 7, 8, 12, 14 and 15 but forbids 5, 9, 10, 13 and 17, as they imply ties. More elegantly, any integer is considered assignable if its binary representation – disregarding any leading zeros – does not contain the substring “01”:

```python
>>> for i in range(17):
...     binary_string = mathtools.integer_to_binary_string(i)
...     is_assignable = mathtools.is_assignable_integer(i)
...     string = '{:} [{}]'.format(i, binary_string, is_assignable)
...     print(string)
...
0: 0 [False]
1: 1 [True]
2: 10 [True]
3: 11 [True]
4: 100 [True]
5: 101 [False]
6: 110 [True]
7: 111 [True]
8: 1000 [True]
9: 1001 [False]
10: 1010 [False]
11: 1011 [False]
12: 1100 [True]
13: 1101 [False]
14: 1110 [True]
15: 1111 [True]
16: 10000 [True]
```

The duration of each container then derives from the product of its prolation and its contents duration – the sum of the durations of its children. Ultimately, all scores derive their durations from the durations of their leaves, prolated as necessary by any tuplets. Recall that containers are mutable. As components are added to and removed from a container, the duration of that container and the offsets of components following the inserted or deleted component adjust dynamically to reflect the altered structure:

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

```text
duration: Duration(1, 1)
```
After inserting an additional four quarter-notes into the staff, the staff reports its duration as doubled, and all of the leaves – both new and old – report their expected start offsets:

```python
>>> staff[2:2] = ['f''4 e''4 d''4 c''4']
>>> show(staff)
```

Likewise, removing the middle two of the previously inserted leaves results in the staff reporting a decreased duration, and all leaves updating their offsets to reflect the deletion:

```python
>>> staff[3:5] = []
>>> show(staff)
```
Finally, multiplier objects may be attached to leaves to multiply their duration. When attached to multi-measure rests, not only does the overall duration of the leaf change, but LilyPond is able to generate typography representing multiple bars of tacet music compressed together:

```plaintext
>>> multimeasure_rest = scoretools.MultimeasureRest(1)
>>> inspect_(multimeasure_rest).get_duration()
Duration(1, 1)
```

```plaintext
>>> show(multimeasure_rest)
```

```plaintext
>>> attach(Multiplier(4), multimeasure_rest)
>>> inspect_(multimeasure_rest).get_duration()
Duration(4, 1)
```

```plaintext
>>> show(multimeasure_rest)
```

### 2.2.4 Named components

Abjad's scores components may be given unique names via their `name` property. Any named component found in the subtree of a given container may be accessed by subscripting the container with its name, regardless of its depth in that container.

Consider the following score, containing a piano staff grouping two staves, each with one voice. Disregard the ugliness of the notation for the time being; it will be cleaned up in section 2.3 and section 2.4:

```plaintext
>>> voice_1 = Voice(name='Voice 1')
>>> voice_1.append(Measure((3, 4), "d'4.. b'16 af'4"))
>>> voice_1.append(Measure((5, 4), "af'4. g'8 c''8 b'4. b'8 r16 d''16"))
>>> upper_staff = Staff(...
...   [voice_1],
...   name='Upper Staff',
... )
>>> voice_2 = Voice(name='Voice 2')
>>> voice_2.append(Measure((3, 4), "r8 c'8 r8 d8 r8 e8"))
>>> voice_2.append(Measure((5, 4), r"r'4 \times 2/3 { g4 fs4 e8 cs8 } r4. fs8")
```
Printing the score as a LilyPond syntax string clearly shows the nested quality of the score hierarchy. Additionally, the previously provided names of some of the containers – e.g. “Voice 1” and “Lower Staff” – also appear in the LilyPond output:

```plaintext
>>> print(format(score))
\new Score <<
\context PianoStaff = "Both Staves" <<
\context Staff = "Upper Staff" {
  \context Voice = "Voice 1" {
    [\time 3/4
d'4..b'16
af'4]
  }
[\time 5/4
af'4.
g'8
c''8
b'4.
b'8
r16
d''16]
}
\context Staff = "Lower Staff" {
  \context Voice = "Voice 2" {
    [\time 3/4
r8
```
Any of the named components within the score hierarchy can be selected by subscripting any container in their proper parentage with their name. For example, the voice container named “Voice 1” can be selected by subscripting the score object with its name, even though it is not immediately contained by the score but is in fact a “great-grandchild” of the score:

```lilypond```
```python
>>> score['Voice 1']
<Voice="Voice 1"(2)>
```
```lilypond```
```python
>>> score['Voice 1'] is voice_1
True
```

Note that the staff group definition above received both a name and a `context_name` keyword argument. While the various voices and staves in the above score all appear in the LilyPond syntax output as \context Staff = "Upper Staff" or \context Voice = "Voice 2", the staff group appears instead as \context PianoStaff = "Both Staves". Its “context name” has been substituted for where StaffGroup would normally appear, allowing Abjad to specify a different LilyPond context for the music it contains.

Some Abjad container classes correspond to LilyPond’s notion of contexts. These include voices, staves, staff groups and scores, but not measures, tuplets or “bare” containers. LilyPond uses contexts during typesetting to maintain various kinds of musical information hierarchically. For example, LilyPond’s Staff context maintains information about the accidentals that have appeared so far in any voice contained by that staff as well as the staff’s
current clef—all of which is necessarily local to a single staff—, while the \textit{Score} context maintains more global information, such as the current tempo and measure number. \texttt{LilyPond} allow composers to define new contexts, and provides a number of specially-defined contexts, e.g. \texttt{ChoirStaff, TabVoice} and \texttt{FiguredBass}. While \texttt{LilyPond}'s contexts may be either named or anonymous, named contexts allow \texttt{LilyPond} to stitch together different sections of music into a single continuous score, allowing different segments of a work to be defined in different files and then concatenated together.

2.3 Indicators

Abjad’s indicators include any object attached to a single score component, such as articulations, textual markup, clefs, tempo or time signature indications. Unlike score components and spanners, indicators do not share a common base class. Instead, they are unified by the means by which they have been attached to components: Abjad’s top-level \texttt{attach()} function. Indicators are generally immutable, like integers and durations. Regardless, in being attached to a score component they do not receive a reference to that component, allowing the same indicator to be attached to many components. Abjad binds the indicator to the component via an \texttt{IndicatorExpression} object, which holds the necessary references to both the indicator and the component along with other important information about the behavior of the attachment.

Consider a simple four note staff. A single accent articulation can be attached to each note in the staff via \texttt{attach():}

\begin{verbatim}
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> accent = Articulation("accent")
>>> for note in staff:
...       attach(accent, note)
...  
>>> show(staff)
\end{verbatim}

Once attached, indicators can be removed via the top-level \texttt{detach()} function. For example, the attached accents can be detached from the last three leaves of the above staff via the \texttt{detach()} function:

\begin{verbatim}
>>> for note in staff[1:]:
...       detach(accent, note)
...  
(Articulation('accent'),)
(Articulation('accent'),)
(Articulation('accent'),)
\end{verbatim}
At this point, only the first note in the staff has anything attached. As with durations and parentage, we can use the component inspector to verify that this is true by testing each note for the existence of an indicator of the class Articulation:

```python
>>> for note in staff:
...    has_articulation = inspect_(note).has_indicator(Articulation)
...    print(note, has_articulation)
...
(Note("c'4"), True)
(Note("d'4"), False)
(Note("e'4"), False)
(Note("f'4"), False)
```

Likewise, we can use the component inspector to retrieve the attached articulation. The inspector provides two methods for retrieving indicators attached to a single component: `get_indicator()` and `get_indicators()`. The latter returns a tuple of zero or more indicators matching any supplied class prototype, while the former returns only one and raises an error if more or less than one indicator matching the supplied class prototype is attached. In the case of retrieving any attached articulations from the staff’s first note – given that there is only one articulation attached –, both methods work perfectly.

```python
>>> inspect_(staff[0]).get_indicators(Articulation)
(Articulation('accent'),)
>>> inspect_(staff[0]).get_indicator(Articulation)
Articulation('accent')
```

Consider again the two-staff score created earlier:

```python
>>> show(score)
```
A variety of indicators can be attached to the leaves in this score to present a more convincing musical result, including clefs, dynamics, tempi and articulations. Note that indicators can be attached to both leaf and container components:

```python
>>> attach(Clef('treble'), score['Upper Staff'])
>>> attach(Clef('bass'), score['Lower Staff'])
>>> lower_leaves = score['Lower Staff'].select_leaves()
>>> for i in [1, 3, 5, 12]:
...     attach(Articulation('staccato'), lower_leaves[i])
...     attach(Tempo((1, 4), 88), score['Voice 1'][0][0])
>>> attach(Dynamic('p'), score['Voice 1'][0][0])
>>> attach(Dynamic('f'), score['Voice 1'][1][0])
>>> attach(Dynamic('ppp'), score['Voice 1'][1][-1])
>>> show(score)
```

2.3.1 Indicator scope

Abjad models how the influence of certain types of indicators persists across components subsequent to the component they attach to via the concept of indicator scope. Indicator scope describes how, for example, all components in a score are governed by one tempo from the moment that tempo appears until the moment a different appears. Likewise, scoping describes how all leaves in a staff are understood to be governed by the staff’s clef up until the point that that clef changes. Different indicators govern different scopes by default. As just described, tempo indications govern the score context, while clef, key signature and dynamics govern the staff context. An indicator governing some component is known as that component’s effective indicator. As with non-scoped indicators, the component inspector can be used to examine if and what indicator of a given type is effective for a given component.

Consider the score above, to which a variety of indicators have just been attached. By inspecting the leaves in both staves with the component inspector’s `get_effective()` method, we can determine what indicators are effective for each leaf. For example, all of the leaves in the upper staff will report that their clef is a treble clef, while all of the leaves in the lower staff will report that theirs is a bass clef:

```python
>>> for leaf in score['Upper Staff'].select_leaves():
...     clef = inspect(leaf).get_effective(Clef)
```
... print(leaf, clef)
...
(\text{Note\("d'4.."\)}, \text{Clef\(name='treble'\)})
(\text{Note\("b'16"\)}, \text{Clef\(name='treble'\)})
(\text{Note\("af'4"\)}, \text{Clef\(name='treble'\)})
(\text{Note\("af'4."\)}, \text{Clef\(name='treble'\)})
(\text{Note\("g'8"\)}, \text{Clef\(name='treble'\)})
(\text{Note\("c'8"\)}, \text{Clef\(name='treble'\)})
(\text{Note\("b'4."\)}, \text{Clef\(name='treble'\)})
(\text{Note\("b'8"\)}, \text{Clef\(name='treble'\)})
(\text{Rest\(\text{'r16'}\)}, \text{Clef\(name='treble'\)})
(\text{Note\("d''16"\)}, \text{Clef\(name='treble'\)})

>>> \text{for \text{leaf} in \text{score[}'Lower Staff'\text{].select_leaves():}
... clef = inspect_\text{(leaf).get_effective(Clef)}
... print(leaf, clef)
...
(\text{Rest\(\text{'r8'}\)}, \text{Clef\(name='bass'\)})
(\text{Note\("c'8"\)}, \text{Clef\(name='bass'\)})
(\text{Rest\(\text{'r8'}\)}, \text{Clef\(name='bass'\)})
(\text{Note\("d'8"\)}, \text{Clef\(name='bass'\)})
(\text{Rest\(\text{'r8'}\)}, \text{Clef\(name='bass'\)})
(\text{Note\("e'8"\)}, \text{Clef\(name='bass'\)})
(\text{Rest\(\text{'r4'}\)}, \text{Clef\(name='bass'\)})
(\text{Note\("g'4"\)}, \text{Clef\(name='bass'\)})
(\text{Note\("fs'4"\)}, \text{Clef\(name='bass'\)})
(\text{Note\("e'8"\)}, \text{Clef\(name='bass'\)})
(\text{Note\("cs'8"\)}, \text{Clef\(name='bass'\)})
(\text{Rest\(\text{'r4.'}\)}, \text{Clef\(name='bass'\)})
(\text{Note\("fs'8"\)}, \text{Clef\(name='bass'\)})

While the tempo indication is only attached to the first leaf in the upper staff, all leaves in the entire score report
that same tempo as their effective tempo:

>>> \text{for \text{leaf in \text{score[}'Upper Staff'\text{].select_leaves():}
... tempo = inspect_\text{(leaf).get_effective(Tempo)}
... print(leaf, tempo)
...
(\text{Note\("d'4.."\)}, \text{Tempo\(\text{duration=Duration\(1, 4\)}, \text{units_per_minute=88\)})
(\text{Note\("b'16"\)}, \text{Tempo\(\text{duration=Duration\(1, 4\)}, \text{units_per_minute=88\)})
(\text{Note\("af'4"\)}, \text{Tempo\(\text{duration=Duration\(1, 4\)}, \text{units_per_minute=88\)})
(\text{Note\("af'4."\)}, \text{Tempo\(\text{duration=Duration\(1, 4\)}, \text{units_per_minute=88\)})
(\text{Note\("g'8"\)}, \text{Tempo\(\text{duration=Duration\(1, 4\)}, \text{units_per_minute=88\)})
(\text{Note\("c'8"\)}, \text{Tempo\(\text{duration=Duration\(1, 4\)}, \text{units_per_minute=88\)})
(\text{Note\("b'8"\)}, \text{Tempo\(\text{duration=Duration\(1, 4\)}, \text{units_per_minute=88\)})
(\text{Rest\(\text{'r16'}\)}, \text{Tempo\(\text{duration=Duration\(1, 4\)}, \text{units_per_minute=88\)})
(\text{Note\("d''16"\)}, \text{Tempo\(\text{duration=Duration\(1, 4\)}, \text{units_per_minute=88\)})

>>> \text{for \text{leaf in \text{score[}'Lower Staff'\text{].select_leaves():}
... tempo = inspect_\text{(leaf).get_effective(Tempo)}
... print(leaf, tempo)
...

36
(Rest('r8'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Note('c"8'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Rest('r8'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Note('d"8'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Rest('r8'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Note('e"8'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Rest('r4'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Note('g4'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Note('fs4'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Note('e8'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Note('cs8'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Rest('r4.'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Note('fs8'), Tempo(duration=Duration(1, 4), units_per_minute=88))

However, the dynamic indications attached to the leaves in the upper staff are effective only for those leaves, and not for the leaves in the lower staff:

>>> for leaf in score['Upper Staff'].select_leaves():
...     dynamic = inspect_(leaf).get_effective(Dynamic)
...     print(leaf, dynamic)
...
(Note("d'4.."), Dynamic(name='p'))
(Note("b'16"), Dynamic(name='p'))
(Note("af'4"), Dynamic(name='p'))
(Note("g'8"), Dynamic(name='f'))
(Note("c'8"), Dynamic(name='f'))
(Note("b'4."), Dynamic(name='f'))
(Note("b'8"), Dynamic(name='f'))
(Rest('r16'), Dynamic(name='f'))
(Note("d'16"), Dynamic(name='ppp'))

Both the clef, tempo and dynamics inspected above made use of their default scope when being attached, specifying implicitly that they be effective at either the staff or score context-level. Indicators can also be attached with an
explicitly-specified scope, overriding any default the indicator’s class might provide. By detaching the implicitly-staff-scoped dynamics, those same dynamic indications can be reattached, explicitly scoped for the “PianoStaff” context, allowing all leaves contained by that piano staff – which includes both the upper and lower staves – to detect their appropriate dynamic level:

```python
>>> piano_dynamic = detach(Dynamic, score['Voice 1'][0][0][0]
>>> forte_dynamic = detach(Dynamic, score['Voice 1'][1][0][0]
>>> pianississimo_dynamic = detach(Dynamic, score['Voice 1'][1][-1][0]
>>> attach(piano_dynamic, score['Voice 1'][0][0], scope='PianoStaff')
>>> attach(forte_dynamic, score['Voice 1'][1][0], scope='PianoStaff')
>>> attach(pianississimo_dynamic, score['Voice 1'][1][-1], scope='PianoStaff')

>>> for leaf in score['Upper Staff'].select_leaves():
...    dynamic = inspect_(leaf).get_effective(Dynamic)
...    print(leaf, dynamic)
...
(Note("d'4.."), Dynamic(name='p'))
(Note("b'16"), Dynamic(name='p'))
(Note("af'4"), Dynamic(name='p'))
(Note("af'4.."), Dynamic(name='f'))
(Note("g'8"), Dynamic(name='f'))
(Note("c'8"), Dynamic(name='f'))
(Note("b'4.."), Dynamic(name='f'))
(Rest('r16'), Dynamic(name='f'))
(Note("d'116"), Dynamic(name='ppp'))

>>> for leaf in score['Lower Staff'].select_leaves():
...    dynamic = inspect_(leaf).get_effective(Dynamic)
...    print(leaf, dynamic)
...
(Rest('r8'), Dynamic(name='p'))
(Note("c'8"), Dynamic(name='p'))
(Rest('r8'), Dynamic(name='p'))
(Note("d8"), Dynamic(name='p'))
(Rest('r8'), Dynamic(name='p'))
(Note("e8"), Dynamic(name='p'))
(Rest('r4'), Dynamic(name='f'))
(Note("g4"), Dynamic(name='f'))
(Note("fs4"), Dynamic(name='f'))
(Note("e8"), Dynamic(name='f'))
(Note("cs8"), Dynamic(name='f'))
(Rest('r4..'), Dynamic(name='f'))
(Note("fs8"), Dynamic(name='f'))
```

Recall the voices in the score were populated by Abjad Measure objects. While LilyPond’s musical model does not make use of explicit measures, Abjad provides measure-like containers as a convenience. When instantiated, Abjad measures attach the appropriate time signature indication to themselves. These indicators are also scoped by default, allowing all of the leaves contained in that measure to detect their appropriate time signature:
>>> measure_1 = score['Voice 1'][0]
>>> measure_2 = score['Voice 1'][1]
>>> inspect_(measure_1).get_indicator(TimeSignature)
TimeSignature((3, 4))

>>> for leaf in measure_1.select_leaves():
...     time_signature = inspect_(leaf).get_effective(TimeSignature)
...     print(leaf, time_signature)
...
(Note("d'4.."), TimeSignature((3, 4)))
(Note("b'16"), TimeSignature((3, 4)))
(Note("af'4"), TimeSignature((3, 4)))

>>> inspect_(measure_2).get_indicator(TimeSignature)
TimeSignature((5, 4))

>>> for leaf in measure_2.select_leaves():
...     time_signature = inspect_(leaf).get_effective(TimeSignature)
...     print(leaf, time_signature)
...
(Note("af'4."), TimeSignature((5, 4)))
(Note("g'8"), TimeSignature((5, 4)))
(Note("c''8"), TimeSignature((5, 4)))
(Note("b'4."), TimeSignature((5, 4)))
(Note("b'8"), TimeSignature((5, 4)))
(Rest("r16"), TimeSignature((5, 4)))
(Note("d''16"), TimeSignature((5, 4)))

2.3.2 Annotations

Indicators may be attached to score components as annotations, “visible” to inspection and potentially scoped, but contributing no formatting to a score hierarchy’s LilyPond output. Additionally, any attached indication which cannot contribute formatting is considered an implicit annotation. Consider some of the indicators used above, such as clefs, time signatures, dynamics and tempo indications. All of these objects can be formatted as LilyPond syntax, and when attached to components in a score will appear in the output if and when the score itself is formatted as LilyPond syntax:

>>> clef = Clef('bass')
>>> print(format(clef, 'lilypond'))
\clef "bass"

>>> time_signature = TimeSignature((S, 4))
>>> print(format(time_signature, 'lilypond'))
\time 5/4

>>> dynamic = Dynamic('p')
>>> print(format(dynamic, 'lilypond'))
\p
tempo = Tempo((1, 4), 88)
print(format(tempo, 'lilypond'))
\tempo 4=88

For example, the above four indicators can be attached to the contents of the following staff which, when formatted, shows the expected format contributions of each indicator. Note that the staff must be wrapped in a score container so that its tempo indication can find the appropriate context:

staff = Staff("g f e d c")
attach(clef, staff)
attach(dynamic, staff)
attach(tempo, staff)
attach(time_signature, staff)
Score([staff])

The above staff can be recreated with the indicators attached as annotations, still effective for each leaf, but providing no format contributions in the output. This results in rather poor notation, falling back on LilyPond's default $\frac{4}{4}$ time signature and treble clef:

staff = Staff("g f e d c")
attach(clef, staff, is_annotation=True)
attach(dynamic, staff, is_annotation=True)
attach(tempo, staff, is_annotation=True)
attach(time_signature, staff, is_annotation=True)
Score([staff])

\new Staff { 
g4
Still, the leaves in this second staff can be inspected and will all report that various indicators attached to the staff, although annotative, are effective for them:

```python
>>> for note in staff:
...    tempo = inspect_(note).get_effective(Tempo)
...    print(note, tempo)
...
(Nota('g4'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Nota('f4'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Nota('e4'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Nota('d4'), Tempo(duration=Duration(1, 4), units_per_minute=88))
(Nota('c4'), Tempo(duration=Duration(1, 4), units_per_minute=88))
```

Annotation allows arbitrary objects to be attached to any score component as a kind of metadata without directly affecting LilyPond’s typesetting, allowing other later compositional processes to inspect or react to those annotations. Abjad provides a variety of indicator classes for this purpose, none of which contribute any formatting, but all of which can be attached and scoped, allowing composers to better model instrumental technique or compositional intent. Some of these annotative indicators include `BowContactPoint`, `IsAtSoundingPitch`, `IsUnpitched`, `StringContactPoint` and `StringTuning`.

2.4 **Spanners**

A final collection of classes, **spanners**, model musical and typographic constructs such as slurs, beams, hairpins, and glissandi which span across different levels of hierarchy in the score tree. Spanners attach to not just one but potentially many components in a score tree, provided those components are contiguous in time and occupy the same logical voice.

A logical voice is a structural relationship between components in a score representing a single musical voice. For example, notes may be contained directly by a staff, without any explicit voice container between those notes and the staff. While those notes do not share an explicit voice, they do share an implicit logical voice. Likewise, notes contained by two explicit voice contexts where the explicit voices share the same name not only inhabit explicit voices but share the same logical voice.
aware of the components, including the leaves, they cover as well as the order in which those components appear. This awareness allows spanners to contribute formatting to a score tree’s LilyPond syntax output conditionally. The first and last leaves covered by a spanner can be formatted specially, and the types of leaves covered by a spanner, their durations or even any indicators attached to those leaves can be considered during formatting, allowing spanners to make use of annotative indicators attached to components.

Consider the following staff, which contains leaves as well as a triplet, itself containing more leaves:

```
>>> staff = Staff(r"c'4 \times 2/3 { c'8 e'8 g'8 } g'2")
>>> show(staff)
```

A variety of spanners can be attached to both the leaves of the staff as well as to the staff itself using Abjad’s attach() function. For example, a slur can be attached to all of the leaves of the staff — including those in the triplet —, a crescendo attached to the staff itself — but not to its leaves —, and two ties attached to the pairs of C- and G-notes which border the bounds of the triplet:

```
>>> slur = Slur()
>>> crescendo = Crescendo()
>>> tie_one = Tie()
>>> tie_two = Tie()
>>> leaves = staff.select_leaves()
>>> attach(slur, leaves)
>>> attach(crescendo, staff)
>>> attach(tie_one, leaves[:2])
>>> attach(tie_two, leaves[-2:])
>>> show(staff)
```

As with indicators, the component inspector provides methods for inspecting any spanners attached to a given component:

```
>>> inspect_(staff[0]).get_spanners()
set([Tie("c'4, c'8"), Slur("c'4, c'8, e'8, g'8, g'2")])

>>> inspect_(staff).get_spanners()
set([Crescendo('<Staff{3}>')])
```
Note above that the results of each inspection for spanner attachments returns a set comprising a unique collection of spanners. Analogous to component parents and children, each component can appear only once in a single spanner’s component selection, and a spanner can only be attached once to the same component. Spanners can also be inspected – much like containers – for any components they cover via indexing, as well as for their duration:

```python
>>> len(slur)
5
```

```python
>>> slur[1]
Note("c'8")
```

```python
>>> slur[:]
Selection(Note("c'4"), Note("c'8"), Note("e'8"), Note("g'8"), Note("g'2"))
```

```python
>>> inspect_(slur).get_duration()
Duration(1, 1)
```

Note that subscripting the spanner with Python’s slice notation [:] returns a selection object containing all of the components to which the spanner attaches. The role of selections is discussed in more detail in section 2.6.

Returning to the graph visualizations of score trees demonstrated earlier will clarify how spanners attach to components. Consider the following new staff, containing two measures of differing durations, each containing a mixture of tuplets and notes. The second measure’s tuplet contains two bare containers, each containing two notes in turn. The leaves of the staff then vary in depth from 2 – contained by a measure, then by the staff – to 4 – contained by a bare container, then a tuplet, a measure and finally the staff:

```python
>>> staff = Staff()
>>> measure_one = Measure((3, 4), r"c'4 \times 2/3 \{ d'4 e'4 f'4 \}")
>>> measure_two = Measure((4, 4), r"\times 3/4 \{ \{ g'4 a'4 \} \{ b'4 c''4 \} \} c'4")
>>> staff.extend([measure_one, measure_two])
>>> show(staff)
```

```
\[\text{\textbf{Noteheads.s2}}\]
\[\text{\textbf{Noteheads.s2}}\]
\[\text{\textbf{3:2}}\]
\[\text{\textbf{4:3}}\]
\[\text{\textbf{4}}\]
\[\text{\textbf{4}}\]
\[\text{\textbf{4}}\]
\[\text{\textbf{4}}\]
\[\text{\textbf{consistent}}\]
\[\text{\textbf{consistent}}\]
\[\text{\textbf{consistent}}\]
\[\text{\textbf{consistent}}\]
```

The staff’s score hierarchy can be visualized as a graph with the leaves arranged at the bottom and grouped together according to their depth:

```python
>>> graph(staff)
```
A Spanner can be attached to the leaves of the staff. Note that while all spanners’ structural behavior is identical to that of their parent Spanner class, Spanner itself provides no formatting beyond the optional typographic overrides discussed in section 2.5, so there is no need to illustrate the newly-altered staff as notation. However, the staff can be graphed again, this time highlighting and connecting together the nodes in the graph representing the components to which the spanner attaches by passing that spanner as a keyword to graph():

```
>>> spanner = spannertools.Spanner()
>>> attach(spanner, staff.select_leaves())
>>> graph(staff, spanner=spanner, graph_attributes={
    ...    'splines': 'curved',
    ...    'concentrate': True,
    ... })
```

Abjad relies on the open-source Graphviz toolkit to visualize graph structures. Abjad also provides an object-oriented model of Graphviz’s input file format as part of its documentation tools subpackage. There are many layout options available when working with Graphviz, and they can be passed as keywords to Abjad’s top-level graph() function, hence the appearance of the graph_attributes keyword. These overrides are necessary to circumvent deficiencies in Graphviz’s handling of this particular graph’s layout.
The spanner can be detached from the staff’s leaves and reattached to the staff itself. Recall that spanners can attach to one or more of any combination of containers and leaves, provided all are contiguous in time and occupy the same logical voice. Note in the following graph that the node representing the staff is both outlined in bold and shaded grey while all of the components underneath it in the score hierarchy are simply shaded grey. In these score hierarchy graphs, only those nodes to which the spanner directly attaches are outlined in bold, while all components that the spanner covers are simply shaded:

```latex
>>> detach(spanner)
>>> attach(spanner, staff)
>>> graph(staff, spanner=spanner)
```

The spanner can then be detached from the staff and attached to the two measures it contains:
The spanner can also be detached again and reattached to a selection created by selecting the contents of both measures and concatenating those selections together. Note that the spanner's components – those nodes outlined in bold and connected by bold arrows – here comprise both leaves and containers, but outline the same collection of leaves as the previous graphs:

Finally, the spanner can be detached from the previous selection and reattached to a new selection comprising the concatenated contents of each measures’ tuplet child:
Yet despite being attachable to both containers and leaves, spanners only contribute formatting to the leaves they cover, not to any container. Why attach spanners to containers at all? Any spanner attached to a container will contribute formatting to the leaves of that container, regardless of what those leaves are, and even if they change during the course of composition. Consider the following small staff example, containing four notes, with the inner two notes wrapped in a bare container:

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

After attaching a slur to the inner container, the slur appears in the notational output, starting on the first leaf of the inner container and stopping on the last leaf:

```python
>>> attach(Slur(), staff[1])
>>> show(staff)
```

More leaves can be inserted into the inner container, both before, between and after the original two notes. The slur still appears starting on the first leaf of the inner container and continuing until its last, although both have changed from the original two notes:
The inner container can be emptied completely, at which point the slur no longer appears in any notational output. However, inspecting the inner container’s inventory of spanners shows that the slur is still attached:

```
>>> staff[1][;] = []
>>> print(format(staff))
\new Staff { 
ocline \note { c'4 } \note { f'4 } }
```

Finally, the inner container can be repopulated, allowing the slur to appear again:

```
>>> staff[1].extend("e'4 d'4")
>>> show(staff)
```

Attaching spanners to containers rather than leaves lets spanners act as a kind of decorator\footnote{2} for those containers in the sense of computer programming design patterns: an object which wraps another object, extending it with new functionality while preserving all of the old. In this case, spanners can extend containers with additional typographic logic, but remain entirely ignorant of the container’s contents.

## 2.5 Typographic overrides

LilyPond’s graphic objects, or grobs in LilyPond parlance, represent all of the typographic glyphs which may appear on the page as the result of the typesetting process. These include note-heads, dots, stems, flags, beams, slurs, accidentals, articulations, clefs, bar-numbers and bar-lines, the lines of the staff itself, as well as all manner of invisible
grobs used to describe the spacing and positioning of other glyphs on the page. Every such grob in LilyPond can be configured extensively. An accidental’s color can be changed, its size increased or decreased, its position relative to its parent grob – the note-head it modifies – adjusted, or its symbol replaced entirely. LilyPond calls the act of modifying graphic objects *overriding*, and the act of returning them to their previous settings *reverting*. Abjad provides a top-level `override()` function for constructing such LilyPond `\override` and `\revert` commands, allowing composers to alter the typography of individual leaves or the contents of entire containers. Similarly to `inspect_()`, calling `override()` on a score component returns an agentive object with that component as its client: a *LilyPond grob name manager*. This manager object exposes an interface for constructing override statements in a dot-chained syntax which mimics the way one would write LilyPond override commands by hand. For example, the style of a single note’s note-head can be changed from the default oval shape to a cross:

```python
>>> note = Note("c'4")
>>> override(note).note_head.style = 'cross'
```

```lilypond
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
```

```plaintext
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
```

In the resulting LilyPond code above, Abjad prepends a `\once` modifier to the override command, obviating the need to explicitly revert the typography override afterward. The same override can be applied against an entire container full of notes, modifying each note in turn:

```python
>>> container = Container("c'4 d'4 e'4 f'4")
>>> override(container).note_head.style = 'cross'
```

```lilypond
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
```

```plaintext
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
```

```plaintext
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
```

```plaintext
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
```

```plaintext
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
```

```plaintext
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
```

```plaintext
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
```

```plaintext
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
```

```plaintext
\once \override NoteHead #'style = #'cross
\once \override NoteHead #'style = #'cross
```
Note above in the LilyPond syntax output that the override command appears just inside the opening brace of the container, and the accompanying revert command appears just before the closing brace. Additionally, much like the earlier spanner examples where a slur was attached directly to a container, overrides applied against a container persist regardless of the contents of that container.

As with score components, composers can also apply typographic overrides against spanners. The initial override commands appear before the first leaf of the spanner – if the spanner covers any leaves at all – and the accompanying revert commands appear following the last leaf of the spanner. Consider the following staff example, containing three measures, with staccato articulations attached to each leaf:

```lilypond
>>> staff = Staff()
>>> staff.append(Measure((3, 4), "c'4 d' r"))
>>> staff.append(Measure((5, 4), r"e'4 \times 2/3 { f' g' r4 } a' b"))
>>> staff.append(Measure((2, 4), "c''8 g' c'4"))
>>> articulation = Articulation('staccato')
>>> for leaf in staff.select_leaves():
...     if isinstance(leaf, Note):
...         attach(articulation, leaf)
...
>>> show(staff)
```

We can override the middle measure to increase the size of the staccato articulations – what LilyPond terms *scripts* – by increasing their font size:

```lilypond
>>> override(staff[1]).script.font_size = 10
>>> show(staff)
```

Next we attach a slur to all leaves starting from the second and going until the next-to-last. We also override that slur to change the style of the note head of each note and chord it covers from the default oval to a cross. Recall that spanners can cross different levels of hierarchy in the score tree. Likewise, typographic overrides applied against spanners also cross container boundaries such as the three measures contained within this staff:
Note the locations of the `\override` and `\revert` commands for both the spanner and the middle measure in the LilyPond syntax output. While the middle measure's `\override` and `\revert` commands appear just within its enclosing braces, the slur's `\override` command appears just before its first leaf – the D quarter-note – and its `\revert` command just after its last leaf – the G eighth-note:

```plaintext
>>> print(format(staff))
\new Staff { 
  { 
    \time 3/4 
    c'4 -\staccato
    \override NoteHead #'style = #'cross
    d'4 -\staccato _ ( 
    r4
  }
  { 
    \override Script #'font-size = #10 
    \time 5/4 
    e'4 -\staccato 
    \times 2/3 { 
      f'4 -\staccato 
      g'4 -\staccato 
      r4
    }
    a'4 -\staccato 
    b4 -\staccato 
    \revert Script #'font-size
  }
  { 
    \time 2/4 
    c''8 -\staccato 
    g'8 -\staccato )
    \revert NoteHead #'style
    c'4 -\staccato
  }
}
```

The typographic overrides of both the slur and the middle measure persist even when changing the contents of that middle measure, such as replacing the triplet with four eighth-notes:
Importantly, typographic overrides *cascade*, with local settings taking precedence over more global settings. Although the previously attached slur overrides the note-head style of all leaves it covers, the note-head styles of leaves within that collection can be overridden themselves. For example, the first, last and next-to-last leaf of the middle measure can be overridden to display a slash symbol rather than cross for their note-heads, effectively overruling the more global typographic commands applied against their covering spanner:

```python
>>> for index in (0, -2, -1):
...     override(staff[1][index]).note_head.style = 'slash'
...
```

By providing an object-oriented interface to the low-level typographic options of its typesetter, LilyPond, Abjad affords composers powerful programmatic control over not only the content but the appearance of the scores they create.

### 2.6 Selecting components

Abjad provides a variety of methods for selecting components from within a score tree. A number of these techniques have already been demonstrated. For example, a score can be subscripted with integers or integer *slices* – e.g. `[1:-1]` – representing the indices of some component or components in order to select those components, as shown in subsection 2.2.1 and later. Similarly, named components can be selected from a score by subscripting that score with their names, as shown in subsection 2.2.4. The leaves at the bottom of a score hierarchy can also be selected en masse via a component’s `select_leaves()` method regardless of their depth within the tree, also shown in subsection 2.2.1. All of these techniques represent *component selection*.

Abjad object-models the concept of component selection explicitly with its `Selection` class. Selections allow arbitrary collection of components to be grouped together and inspected without altering the score tree in the
act of selecting. Unlike containers or spanner, components grouped by a selection have no reference back to that selection. Such unidirectional references prevent considerable complexity in the system and allow selections to be made and discarded with ease. However, like containers, spanners and the component inspector, selections expose a variety of methods for examining the components they group. Consider the following staff, which contains a variety of spanners attached to its leaves:

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

A selection can be made simply by subscripting the staff to select from the third leaf up to, but not including, the seventh:

```python
>>> selection = staff[2:6]
>>> print(format(selection))
selectiontools.SliceSelection(
    (scoretools.Note("e'8"), scoretools.Note("f'8 "), scoretools.Note("g'8 ["), scoretools.Note("a'8"),)
)
```

That selection can be iterated over like a list, and inspected for its total duration and timespan – its start and stop offsets, discussed at length in chapter 3:

```python
>>> for component in selection: ... component ...
... Note("e'8")
... Note("f'8")
... Note("g'8")
... Note("a'8")

>>> selection.get_duration()
Duration(1, 2)

>>> selection.get_timespan()
Timespan(start_offset=Offset(1, 4), stop_offset=Offset(3, 4))
```

Furthermore, the selection can be passed as the target to which a spanner attaches. Recall the use of subscripts in calls to `attach()` in section 2.4. All of those container subscripting expressions produced selection objects, although they were not displayed on the console at any point. In this case, we can attach a glissando spanner to the leaf selection:
Selections can also be queried for the spanners covering the components they contain:

```python
>>> for spanner in selection.get_spanners():
...    spanner
...   Glissando("e'8, f'8, g'8, a'8")
   Beam("g'8, a'8, b'8, c''8")
   Beam("c'8, d'8, e'8, f'8")
   Slur("c'8, d'8, e'8, f'8, g'8, a'8, b'8, c''8")
```

### 2.6.1 Logical ties

Abjad provides a number of selection subclasses, some of which have already been described. For example, the `Parentage` class, discussed in subsection 2.2.2 is actually a type of selection, and affords the same methods described above in addition to many others. One of the most fundamental selection subclasses in Abjad is the `LogicalTie`. Logical ties – which are distinct from tie spanners – model the concept of one or more leaves tied together to create a single attack point with an aggregate duration. Logical ties containing a single note or chord are termed *trivial*. Logical ties comprising rests are not well-defined, but can certainly be instantiated like any other selection if a composer would find them useful.

Consider the following staff, which contains a mixture of notes and rests. The “~” symbol in LilyPond syntax indicates that two notes should be joined by a tie. Notably, this staff contains notes which are untied, notes which are tied to a single other note, a chain of three notes tied together, and ties which cross the bounds of a triplet:

```python
>>> staff = Staff(context_name='RhythmicStaff')
>>> staff.extend(r"c'4 r4.. c'16 ~ c'4 ~ \times 2/3 \{ c'8 r4 c'8 ~ \} c'4 r4 c'4")
>>> show(staff)
```

The most direct way to select a leaf’s logical tie is via the component inspector’s `get_logical_tie()` method. Once selected, the logical tie can be examined or printed much like any other selection:

```python
>>> logical_tie = inspect_(staff[2]).get_logical_tie()
>>> print(format(logical_tie))
selectiontools.LogicalTie(54
```
Like `Parentage`, logical tie selections expose properties which are specific to their domain. Two of the most important such properties are a logical tie’s *head* and *tail*: the first and last leaves of the logical tie respectively. We can label the previously selected logical tie’s head and tail with textual *markup*:

```python
>>> logical_tie.head
Note("c'16")

>>> logical_tie.tail
Note("c'8")
```

```python
>>> head_markup = Markup('head', direction=Up)
>>> head_markup = head_markup.pad_around(0.5).box().pad_around(0.5)
>>> tail_markup = Markup('tail', direction=Up)
>>> tail_markup = tail_markup.pad_around(0.5).box().pad_around(0.5)
>>> attach(head_markup, logical_tie.head)
>>> attach(tail_markup, logical_tie.tail)
>>> show(staff)
```

Logical ties expose other pertinent properties. For example, a logical tie can be queried for whether all of its leaves occupy the same parent, whether none of its leaves are rests – therefore the logical tie is pitched –, or whether the logical tie is trivial:

```python
>>> logical_tie.all_leaves_are_in_same_parent
False

>>> logical_tie.is_pitched
True
```

LilyPond provides a *markup* environment for formatting text and graphics, which can then be attached to leaves in the score, used as titles, headers, footers and so forth, or set as the *stencil* for various grobs, overriding that grob’s original appearance entirely. Abjad models LilyPond’s markup environment with its `Markup` class, which exposes many dot-chainable methods for configuring markup objects, mirroring the markup functions available in LilyPond.

---

8LilyPond provides a *markup* environment for formatting text and graphics, which can then be attached to leaves in the score, used as titles, headers, footers and so forth, or set as the *stencil* for various grobs, overriding that grob’s original appearance entirely. Abjad models LilyPond’s markup environment with its `Markup` class, which exposes many dot-chainable methods for configuring markup objects, mirroring the markup functions available in LilyPond.
Trivial logical ties can also be selected. For example, the first leaf in the above staff is covered by no tie spanner at all. Still, that leaf can be modeled as a logical tie containing only a single note. A trivial logical tie’s head and tail are then necessarily the same component:

```python
>>> trivial_logical_tie = inspect_(staff[0]).get_logical_tie()
>>> trivial_logical_tie.is_trivial
True
```

```python
>>> print(format(trivial_logical_tie))
selectiontools.LogicalTie(
  ( scoretools.Note("c'4"), ),
)
```

```python
>>> attach(head_markup, trivial_logical_tie.head)
>>> attach(tail_markup, trivial_logical_tie.tail)
>>> show(staff)
```

2.6.2 Iteration

Abjad provides a variety of techniques for iterating over the components in a score tree. The “simplest” iteration technique employed in Abjad relies on Python’s iteration protocol, which allows objects to be iterated over in “for” loops – as well as many other constructs – yielding up items from their contents one at a time:

```python
>>> for x in ['foo', 10, 3.14159]:
...    x
...    'foo'
...    10
...    3.14159
```

Recall the two-staff score created earlier. Using this basic iteration technique – already demonstrated in subsection 2.2.1 –, the components immediately contained by any Abjad container can be iterated over as though that container were a list. For example, the components immediately contained in the score, the score’s staff group and the voice contained in the upper staff can be iterated over:
Abjad also provides an iteration agent – much like Abjad’s component inspector – which exposes a variety of iteration methods. Calling the top-level iterate() function against a score component returns an iteration agent configured to iterate over the contents of that component:

```python
>>> iteration_agent = iterate(score)
```

Perhaps the most fundamental method of score tree traversal is depth-first iteration\(^7\) whereby not only is a given container iterated over but each container it contains as well and each container those contain, recursively down to the leaves of the score tree. The iteration agent exposes depth first traversal via a method call. We can demonstrate this traversal algorithm by inspecting the parentage of each score component yielded during each step of the traversal process and printing them to the terminal indented by their “depth” relative to the root of the score tree:

```python
>>> for component in iteration_agent.depth_first():
...    parentage = inspect_(component).get_parentage()
...    component_depth = parentage.depth
...    indent = ' ' * component_depth
...    string = '{0}'.format(indent, repr(component))
...    print(string)
...<Score<<1>>>
```
Note how the above indented output mirrors the graph visualization of the same score tree, with the indent of each components representation corresponding to the number of edges between that component and the score tree's root score container:

```python
>>> graph(score)
```
In fact, Abjad’s score tree graph visualizations, like many of the other more-specialized score tree iteration techniques – including the select_leaves() method shown throughout this document –, rely on this depth-first traversal. One of the most common iteration techniques based closely on depth-first traversal is iteration by class, whereby a score tree is iterated over recursively and only those items matching a component class prototype – e.g. only rests, only tuplets, only notes and chords, or any type of leaf – are yielded. We can demonstrate this technique by iterating over the same score by rests and attaching some markup to each yielded rest:\footnote{The built-in Python \texttt{enumerate} function returns an iterator which yields pairs of indices and items from its iterable argument, effectively giving the index of each item in that original iterable. In the case of the following code example, the original iterable may itself be in iterator object rather than a list or tuple. This is an example of Python’s iterator protocol in action, as \texttt{enumerate} accepts any argument as long as it can be iterated over.}

```python
>>> iterator = iteration_agent.by_class(Rest)
>>> for count, rest in enumerate(iterator, start=1):
...     string = 'R:{}'.format(count)
...     markup = Markup(string, direction=Up)
...     markup = markup.pad_around(0.5).box().pad_around(0.25)
...     attach(markup, rest)
... 
>>> show(score)
```

Note how the attached markup orders rests from left-to-right, starting in the upper staff and then proceeding to the lower staff. Such a traversal mirrors the order in which those rests appear during depth-first traversal. Now, before
proceeding to a demonstration of the next technique, we first iterate over all of the components in the score tree depth-first and detach any markup previously attached, such as the rest-numbering markup attached above:

```python
>>> for component in iteration_agent.depth_first():
...     detached = detach(Markup, component)
```

Iteration by *timeline* is another important score traversal technique, yielding each component – optionally filtered by a class prototype – in the order in which they in time relative to the start of the score. Components with identical start times are yielded according to their score index – a tuple of indices describing the index of each of the components in a component’s parentage in *their* parents. We can demonstrate this iteration technique by iterating over all of the leaves in time-wise order, attaching markup to display their index in the timeline:

```python
>>> iterator = iteration_agent.by_timeline(scoretools.Leaf)
>>> for count, leaf in enumerate(iterator, 1):
...     string = 'T:{:}'.format(count)
...     markup = Markup(string, direction='Up')
...     markup = markup.pad_around(0.5).box().pad_around(0.25)
...     attach(markup, leaf)
... >>> show(score)
```

Composers are often concerned with contiguous *runs* of notes and chords rather than any other collection of components. The iteration agent’s `by_run()` method iterates over such contiguous groups by class:

```python
>>> lower_leaves = score['Voice 2'].select_leaves()
>>> for run in iterate(lower_leaves).by_run(Note):
...     run
...     (Note("c'8"),)
     (Note('d8'),)
     (Note('e8'),)
     (Note('g4'), Note('fs4'), Note('e8'), Note('cs8'))
     (Note('fs8'),)
```
Finally, score trees can be traversed by logical ties – selections representing one or more component tied together by a tie spanner, as described in section 2.6. Before demonstrating this technique we must both attach ties to some of the leaves in the upper staff – the A-flats and the Bs – and strip out the previously-attached markup:

>>> attach(Tie(), score['Upper Staff'].select_leaves()[2:4])
>>> attach(Tie(), score['Upper Staff'].select_leaves()[6:8])
>>> for component in iterate(score).depth_first():
...    detached = detach(Markup, component)
...

With the score prepared, all pitched logical ties in the upper staff can be iterated over and markup attached to the head of each:

>>> upper_leaves = score['Voice 1'].select_leaves()
>>> for logical_tie in iterate(upper_leaves).by_logical_tie(pitched=True):
...    markup = Markup('H', direction=Up)
...    markup = markup.pad_around(0.5).box()
...    attach(markup, logical_tie.head)
...>

2.6.3 Selectors

Component selectors are highly-configuration objects which object-model the act of selecting components – as demonstrated above – by aggregating together a series of small callback classes into a pipeline. Each callback describes one step in the act of performing a complex selection, such as selecting leaves, selecting logical ties, selecting the first item from a selection, or selecting items whose length is longer than some count. Selectors allow composers to model the procedure by which they, for example, select rest-delimited runs of notes and chords, then select all logical ties within those runs and finally select the last leaf of each logical tie, even if that logical tie contains only a single leaf. Such a procedure, once codified in a fully-configured selector, can be applied against any leaf, container or selection of components.

When called against a component, an unconfigured selector simply selects that component, returning the selection wrapped within a sequence so that the result can be iterated over:
The selector can be further configured by calling one of its various generating methods. These methods duplicate the selector's current sequence of callbacks and append a new one, returning a new selector containing the modified callback sequence. For example, the previously-created selector can be configured to select the leaves in each selection it processes:

```python
>>> selector = selector.by_leaves(flatten=True)
>>> for x in selector(score['Upper Staff']):
...   x
...
<Staff"Upper Staff"[1]>
```

The selector's storage format shows its callback sequence. The leaf selection callback appears here as a `PrototypeSelectorCallback`, itself configured to select components of type `Leaf`:

```python
>>> print(format(selector))
selectortools.Selector(
   callbacks=(
     selectortools.PrototypeSelectorCallback(
       prototype=scoretools.Leaf,
       flatten=True,
     ),
   ),
)
```

The selector can be configured to select all pitched logical-ties from within the previous selections.

```python
>>> selector = selector.by_logical_tie(pitched=True)
>>> for x in selector(score['Upper Staff']):
...
```

---

10 Selector configuration is conceptually similar to the textual markup configuration used elsewhere in this chapter. In both cases, the object can made to produce new, modified versions of itself through the use of generative methods.
Again, note the addition of a new callback to the selector’s callback pipeline:

```python
>>> print(format(selector))
selectortools.Selector(
    callbacks=(
        selectortools.PrototypeSelectorCallback(
            prototype=scoretools.Leaf,
            flatten=True,
        ),
        selectortools.LogicalTieSelectorCallback(
            flatten=True,
            pitched=True,
            trivial=True,
        ),
    ),
)
```

The selector can be configured to filter out logical ties shorter than a duration of \(\frac{1}{4}\):

```python
>>> selector = selector.by_duration('>', Duration(1, 4))
```

```python
>>> for x in selector(score['Upper Staff']):
...    x
... LogicalTie(Note("d'4.."),)
LogicalTie(Note("af'4"), Note("af'4."))
LogicalTie(Note("b'4."), Note("b'8"))
```

The first item from each selected logical-tie can then be selected by using Python’s index subscripting syntax.

```python
>>> selector = selector[0]
>>> for x in selector(score['Upper Staff']):
...    print(repr(x))
...    ContiguousSelection(Note("d'4.."),)
ContiguousSelection(Note("af'4"),)
ContiguousSelection(Note("b'4."),)
```

Note that the returned selections each contain a single note. This is not necessarily useful yet. By appending a single additional flattening callback to the selector’s callback stack, the selector can be configured to return a flat selection of components which can be iterated over, yielding only a single note on each iteration:
Unlike component iterators – which eventually terminate –, selectors can be re-used. The same selector which was called against the upper staff in the above example can be called again, returning the same result. As with the examples in subsection 2.6.2, we can label each component returned by the selector with some markup. Because the score was already labeled – marking the head of each logical tie in the upper staff – we first remove all markup by iterating depth-first:

Printing the storage format of the current selector shows its deeply-nested configuration:
Selectors are incredibly flexible. They allow composers to describe in a great degree of detail the process they wish
to use to select components from a score for further transformation or ornamentation. As will be elaborated on in
the following chapters, they are also useful because they are fully object-modeled as classes. Such object-modeling
allows them to not only be used to configure still-larger aggregate compositional objects, but also be persisted to
disk due to their well-formed storage format.

2.7 Templatting & persistence

One of the last, but most important aspects of working with Abjad does not concern modeling notation at all.
The act of templatting takes one object and replaces the values assigned to one or more of its properties with new
values, returning a new object based on – but differing from – the old, while the old object remains unchanged.
Templatting’s combination of copying and re-configuration is a standard concepts in computer science, but needs
to be introduced here due to its pervasive use throughout the rest of this document.

Recall the earlier “rhythm-maker” code example from section 2.1:

```python
>>> rhythm_maker = rhythmmakertools.TaleaRhythmMaker(
    ... talea=rhythmmakertools.Talea([1, 2, 3], 16),
    ... extra_counts_per_division=(1, 0, 2, 1, 0),
    ... output_masks=[
    ...     rhythmmakertools.SustainMask([1, 3]),
    ...     rhythmmakertools.SilenceMask([-1]),
    ...     rhythmmakertools.NullMask([0]),
    ... ],
    ... tie_specifier=rhythmmakertools.TieSpecifier(
    ...     tie_across_divisions=True,
    ... ),
    ... )
>>> print(format(rhythm_maker))
rhythmmakertools.TaleaRhythmMaker(
    talea=rhythmmakertools.Talea(
        counts=(1, 2, 3),
        ... )
    ... )
```
Without yet discussing the structure or purpose of this particular object – all of which is covered at length in chapter 3 and chapter 4, especially in section 3.5 – we can demonstrate Abjad’s templating functionality via its top-level new() function. For example, new() can be used to template a new rhythm-maker from the old one shown above, replacing the value referenced by one of its properties with a new value. Here we replace its extra_counts_per_division with the sequence \([1, 2, 3]\). Note the change in the new rhythm-maker’s storage format:
Multiple values can be replaced, and previously unspecified values specified. Here we replace both the new rhythm-makers's `talea` and `extra_counts_per_division` values and specify a previously unspecified `beam_specifier`:

```python
>>> new_rhythm_maker = new(new_rhythm_maker,
...    talea=rhythmmakertools.Talea([2, 1], 8),
...    extra_counts_per_division=[1, 0, 1, 2, 0, 1],
...    beam_specifier=rhythmmakertools(BeamSpecifier(
...        beam_each_division=True,
...        beam_divisions_together=True,
...        use_feather_beams=False,
...    ),
... ),
... )
>>> print(format(new_rhythm_maker))
rhythmmakertools.TaleaRhythmMaker(
    talea=rhythmmakertools.Talea(
        counts=(2, 1),
        denominator=8,
    ),
    extra_counts_per_division=(1, 0, 1, 2, 0, 1),
    beam_specifier=rhythmmakertoolsBeamSpecifier(
        beam_each_division=True,
        beam_divisions_together=True,
        use_feather_beams=False,
    ),
    output_masks=rhythmmakertools.BooleanPatternInventory(
        (rhythmmakertools.SustainMask(
            indices=(1,),
            invert=True,
        ),
        rhythmmakertools.SilenceMask(
            indices=(-1,),
        ),
        rhythmmakertools.NullMask(
            indices=(0,),
        ),
    ),
    tie_specifier=rhythmmakertools.TieSpecifier(
        tie_across_divisions=True,
    ),
)
```

Arbitrarily-nested values can also be replaced by specifying names delimited by double underscores. Here the denominator of the `talea` can be reconfigured directly, changing it from the previous value of 8 to 4:
>>> new_rhythm_maker = new(new_rhythm_maker,
...   ...   talea__denominator=4,
...   ...   )
>>> print(format(new_rhythm_maker))
rhythmmakertools.TaleaRhythmMaker(
   talea=rhythmmakertools.Talea(
       counts=(2, 1),
       denominator=4,
   ),
   extra_counts_per_division=(1, 0, 1, 2, 0, 1),
   beamSpecifier=rhythmmakertools.BeamSpecifier(
       beam_each_division=True,
       beam_divisions_together=True,
       use_feather_beams=False,
   ),
   output_masks=rhythmmakertools.BooleanPatternInventory(
       (   
           rhythmmakertools.SustainMask(
               indices=(1,),
               invert=True,
           ),
           rhythmmakertools.SilenceMask(
               indices=(-1,),
           ),
           rhythmmakertools.NullMask(
               indices=(0,),
           ),
       ),
   ),
   tieSpecifier=rhythmmakertools.TieSpecifier(
       tie_across_divisions=True,
   ),
)

Storage formatting is used throughout Abjad and tools extending Abjad for a number of reasons. For one, it provides a full description of a potentially complex object’s configuration in as human-readable a format as possible. Storage formats are deterministic; any storage-formattable object with a given configuration is guaranteed to format in one and only one way. Non-trivial storage formats also take up multiple lines, making them well-suited to line-based version control systems, such as Subversion, Git and Mercurial – among many others –, discussed in subsection 5.3.2. Most importantly, the storage format for any storage-formattable object is guaranteed to be evaluable as code in Abjad’s namespace. Any object returned by evaluating another object’s storage format as a string will be configured identically to the original and compare equally. This is a kind of human-readable serialization. For example, the selector created previously in subsection 2.6.3 can be persisted via its storage format and recreated by evaluating that storage format in Abjad’s namespace. The new selector compares equally to the old:
>>> print(format(selector))
selectortools.Selector(
    callbacks=(
        selectortools.PrototypeSelectorCallback(
            prototype=scoretools.Leaf,
            flatten=True,
        ),
        selectortools.LogicalTieSelectorCallback(
            flatten=True,
            pitched=True,
            trivial=True,
        ),
        selectortools.DurationSelectorCallback(
            duration=selectortools.DurationInequality(
                operator_string='>,'
                duration=durationtools.Duration(1, 4),
            ),
        ),
        selectortools.ItemSelectorCallback(
            item=0,
            apply_to_each=True,
        ),
        selectortools.FlattenSelectorCallback(
            depth=-1,
        ),
    ),
)

>>> selector_format = format(selector)
>>> evaluated_selector = eval(selector_format)
>>> evaluated_selector == selector
True

Importantly, storage formatting affords persistence to disk. Abjad’s persistence agent – conceptually similar to the inspection agent, iteration agent and others – can write any storage-formattable object to disk as an importable Python module, automatically including the appropriate library import statements at the top of the file.

>>> persistence_agent = persist(selector)
>>> persistence_agent.as_module(
...     module_file_path='a_selector_module.py',
...     object_name='persisted_selector',
... )

The persisted module file can then be read. Note the various import statements at the top – durationtools, scoretools, selectortools – which have been determined by introspecting the selector:

>>> with open('a_selector_module.py', 'r') as module_file_pointer:
...     module_file_contents = module_file_pointer.read()
from abjad.tools import durationtools
from abjad.tools import scoretools
from abjad.tools import selectortools

persisted_selector = selectortools.Selector(
    callbacks=(
        selectortools.PrototypeSelectorCallback(
            prototype=scoretools.Leaf,
            flatten=True,
        ),
        selectortools.LogicalTieSelectorCallback(
            flatten=True,
            pitched=True,
            trivial=True,
        ),
        selectortools.DurationSelectorCallback(
            duration=selectortools.DurationInequality(
                operator_string='>',
                duration=durationtools.Duration(1, 4),
            ),
        ),
        selectortools.ItemSelectorCallback(
            item=0,
            apply_to_each=True,
        ),
        selectortools.FlattenSelectorCallback(
            depth=-1,
        ),
    ),
)

The persisted selector can also be imported from the persisted module, like any other importable Python code. It too compares equally to the original selector:

from a_selector_module import persisted_selector
persisted_selector == selector
True

Storage-formatting, templating and persistence are crucial for working with Abjad at a high-level. Many of the compositionally-interesting devices afforded by Abjad – objects which describe compositional processes, transformations, factories and the like – involve a considerable amount of configuration. Templating affords variation in these situations, allowing composers to create new objects from old, varying as few or as many aspects of the object under consideration as they wish, but saving them from the work of having to define a new object from scratch. Likewise, disk persistence allows composers to easily save their work.
2.8 Other tools

Abjad provides many other tools for working with notation and modeling musical concepts, too many to discuss in full here. A wide variety of Abjad’s classes for modeling time, rhythm and meter are explored in chapter 3, along with various extensions implemented in the Consort composition-modeling library, but others bear mentioning now.

2.8.1 Mutation

Abjad’s mutation agent – similar again its inspection, iteration and persistence agents – affords techniques for destructively transforming already-instantiated score objects. Accessible via the top-level `mutate()` function, the mutation agent can split, transpose and replace components within a score. For example, the leaves in a score can be split cyclically, every other split shard transposed by a minor-second, and slurs and articulations attached to each shard:

```python
>>> staff = Staff("c'4 d'4 e'4 f'4 g'4 a'4 b'4 c''4")
>>> agent = mutate(staff[:])
>>> for i, shard in enumerate(agent.split(durations=[(3, 16)], cyclic=True)):
...     if i % 2:
...         mutate(shard).transpose('m2')
...     if 1 < len(shard):
...         attach(Slur(), shard)
...         attach(Articulation('accent'), shard[0])
...     ...
>>> show(staff)
```

2.8.2 Pitch modeling

Abjad provides a rich model of pitch, making explicit distinctions between named pitches – C4, Db5 – and numbered pitches – 0, 13 – as well as providing concrete models of named and numbered pitch-classes, intervals, interval-classes, pitch-ranges, octaves, accidentals, and all manner of specialized collection classes for these pitch objects, including pitch sets and interval-class vectors, each with transformation methods pertinent to their domain.

```python
>>> pitch_segment = pitchtools.PitchSegment("c' d' e' fs' gf'")
>>> pitch_segment
PitchSegment(["c'", "d'", "e'", "fs'", "gf'"])
>>> pitch_segment = pitch_segment.rotate(1, transpose=True)
>>> pitch_segment
PitchSegment(["c'", "fs'", "gs", "as'", "bs'")
```
Notably, Abjad’s note-head objects – aggregated into its notes and chords – always store their pitch information as named pitches, guaranteeing an internal representation as close to the notational output as possible.

2.8.3 Parsers

Abjad comes equipped with a number of parsers. Most notably, Abjad implements a parser for LilyPond’s syntax, and can understand much – although not all – of LilyPond’s grammar. Any time a note, rest, chord, container or even textual markup is instantiated with a string, the LilyPond parser is at work. The top-level `parse()` function provides access to the this parser:

```python
>>> string = r"\new Staff { c'4 \p \< d'4 e'4 f'4 } \! }"
>>> result = parse(string)
>>> show(result)
```

The LilyPond parser can also be instantiated by hand and called:

```python
>>> string = r"\new Staff { c' _ \markup { X \bold Y \italic Z "a b c" } }"
>>> parser = lilypondparsertools.LilyPondParser()
>>> result = parser(string)
>>> show(result)
```

Abjad also implements a simple parser for RTM-like *rhythm-tree* syntax which parses strings into rhythm-tree objects which can then be converted into proper tuplets:

```python
>>> rtm_parser = rhythmstreamtools.RhythmTreeParser()
>>> rtm_syntax = '(1 (1 (2 (1 1) -1) -2))'
>>> rtm_container = rtm_parser(rtm_syntax)[0]
>>> abjad_container = rtm_container(Duration(1, 2))[0]
>>> show(abjad_container)
```
2.8.4 Notation factories

Composers can certainly create every object in a score by hand, instantiating each note, chord or measure one-at-a-time. However, with experience, when one is already working with notation computationally one tends to want means of generating many score objects at once. Abjad provides a number of factory classes and functions for generating arbitrarily large amounts of notation. The simplest such factory is the `scoretools.make_notes()` function, which combines sequences of pitches and durations in a patterned way to create a selection of leaves:

```python
>>> pitches = ['c''', 'ef''', 'g''', 'b']
>>> durations = [(3, 8), (5, 16), (1, 4)] * 3
>>> notes = scoretools.make_notes(pitches, durations)
>>> staff = Staff(notes)
>>> show(staff)
```

Abjad’s `rhythm-maker` family of classes – discussed in depth in section 3.5 – provide a collection of highly-customizable factories for generating rhythms:

```python
>>> rhythm_maker = rhythmmakertools.TaleaRhythmMaker(
...    talea=rhythmmakertools.Talea(
...      counts=[1, 2, 3, 4],
...      denominator=16,
...    ),
...    tie_specifier=rhythmmakertools.TieSpecifier(
...      tie_across_divisions=True,
...    ),
...)
>>> divisions = [(3, 8), (5, 8), (2, 8), (3, 8), (3, 8)]
>>> rhythm = rhythm_maker(divisions)
>>> staff = Staff(rhythm)
>>> show(staff)
```

Abjad’s `score template` classes act as factories for scores, fully populated with voices, staves, staff groups, and potentially clefs and instruments, but devoid of any “count-time” components – notes, rests, chords, tuplets or measures
– which comprise musical content. Score templates play a large role in the work described in later chapters, especially chapter 4, as they provide a way for multiple score-generating processes to synchronize on a common score structure. For example, `templatetools` provides a score template class for generating string quartet scores:

```python
>>> score_template = templatetools.StringQuartetScoreTemplate()
>>> score = score_template()
>>> print(format(score))
```

2.9 Conclusion

This chapter discussed the core components and concepts for working with notation in Abjad. Composers aggregate score components, consisting of leaves and containers, into nested, hierarchical score structures. Indicators and spanners attach to the components in the tree, adding additional structuration, annotation or typographic embellishment. Iteration and selection then provide means for examining the constructed score tree, allowing composers to locate components within the growing composition for examination, mutation or decoration. The chapters
that follow build on these techniques, and describe increasingly high-level means of organizing time and generating notation en masse with Abjad.
 Consort is an open-source Python package extending Abjad and implementing a model of composition which relies on a number of interrelated but distinct approaches to working with musical time. For example, Abjad’s `timespans` suggests a “coarse” approach to musical time. These objects represent arbitrary durated events on a timeline, without respect for score hierarchy or meter. They are well-suited for modeling large-scale phrasing and gestural density structures and can be transformed through splitting, scaling, stretching, and translation among various other manipulations. Because they have no notational reality, they can model temporal concepts unsuited to notated music – without introducing additional complexity—, such as arbitrarily overlapping events. Timespans may also be annotated, allowing composers to position metadata anywhere on a timeline, much like arranging audio regions in a DAW. Moreover, every durated object in a score hierarchy can be described as a timespan, allowing score components to engage in abstract time relations. In contrast, no-
tated rhythms, composed of note, rest, tie and tuplet objects – among others–, provide the most “fine-grained” approach to musical time. While incredibly expressive, fully notated rhythms are potentially complex to create, and must ultimately be anchored in a score hierarchy. Highly-configurable *rhythm-maker* objects ameliorate the complexity of creating notated rhythms by providing a high-level interface to the process of rhythm generation. Abjad’s hierarchical model of *meter* coordinates time and rhythm vertically across different levels of depth in the score tree, and bridges Consort’s coarse and fine stages of rhythmic interpretation. Meter sequences can be generated from timespan-based phrase structures, and those meter sequences used to transform notated rhythms in turn. A thorough discussion of the implementation of these time models and their implications will clarify an analysis in chapter 4 of Consort’s *score interpretation stage*.

### 3.1 Timespans

*Timespans* model left-closed / right-open intervals of time positioned absolutely along a timeline. Every timespan describes a range of offsets $x$ starting with – and including – some start offset $A$ and leading up to – but not including – some stop offset $B$:

$$A \leq x < B$$ (3.1)

Note that all leaves in a score describe such half-bounded intervals of time. Adjacent notes in a score do not overlap but rather abut one another because their timespans do not overlap. In fact, every durated object in a score – every note, chord, rest, measure, staff, and even the score itself – can be described in terms of timespans. Yet, while score objects can always be expressed as timespans, those timespans themselves do not – by definition – refer to any score objects. Abjad implements timespans as immutable constants, much like Abjad’s *Pitch* and *Duration* objects, and similarly to Python’s implementation of numbers and strings. Constancy allows timespans to avoid a variety of computational reference problems. For example, multiple objects can reference the same timespan without fear of that timespan changing state, much as multiple objects can reference the integer 11 without fear that that same integer will change into the integer 5.

Abjad implements timespans via the *Timespan* class in its *timespantools* library. The following code shows the definition of a timespan object beginning at the offset $\frac{1}{4}$ and continues up until the offset $\frac{3}{2}$:

```python
>>> timespan = timespantools.Timespan(
...     ...     start_offset=Offset(1, 4),
```
Like many objects in Abjad, timespan can be formatted for human-readable textual inspection, or displayed as a graphic illustration:

```
>>> print(format(timespan))
timesteptools.Timespan(
    start_offset=durationtools.Offset(1, 4),
    stop_offset=durationtools.Offset(3, 2),
)
```

```
>>> show(timespan)
```

The `Timespan` class provides a large number of methods and properties for inspecting timespans, comparing them to other timespans or offsets, and for operating on timespans to generate new timespans. Once instantiated, a timespan can be examined for its start offset, stop offset and duration. Because of the `Timespan` class’ immutability, these properties are read-only and therefore can only be accessed, but not changed:

```
>>> timespan.start_offset
Offset(1, 4)
```

```
>>> timespan.stop_offset
Offset(3, 2)
```

```
>>> timespan.duration
Duration(5, 4)
```

A timespan’s start offset must be equal to or less than its stop offset. Timespans with identical start and stop offsets have a duration of 0 and effectively model a single time-point. Such timespans are considered not “well-formed”:

```
>>> timepoint_timespan = timesteptools.Timespan(1, 1)
```

```
>>> timepoint_timespan.duration
Duration(0, 1)
```

```
>>> timepoint_timespan.is_well_formed
False
```

Both the `start_offset` and `stop_offset` keywords to the `Timespan` class’ initializer are optional, and default to Abjad’s built-in rational constants `NegativeInfinity` and `Infinity` respectively. A timespan created without specifying either a start or stop offset effectively describes the timespan which encompasses all possible offsets in time:
By specifying only a start or stop offset, timespans can also be created which encompass the infinite set of offsets up until some stop offset, or the infinite set of offsets starting at and following some start-offset:

```python
>>> timespantools.Timespan(stop_offset=0)
Timespan(start_offset=NegativeInfinity, stop_offset=Offset(0, 1))
```

Timespan objects also partake in Abjad's templating regime. New timespans can be created from old ones through the use of the top-level `new()` function:

```python
>>> new(timespan, stop_offset=(5, 16))
Timespan(start_offset=Offset(1, 4), stop_offset=Offset(5, 16))
```

Note that the above timespans have been configured from input in various formats, such as explicit `Offset` objects, integers, and even numerator/denominator pairs. Timespans always coerce arguments given for their `start_offset` and `stop_offset` properties into explicit `Offset` instances. Abjad's other classes and functions implement similar offset, duration, and even pitch-coercion behavior pervasively.

### 3.1.1 Annotated timespans

While the `Timespan` class only has two configurable properties – its start offset and stop offset –, subclassing allows for the creation of new classes with the same core functionality as `Timespan` but extended to support new behaviors and configurations. As an example, Abjad's `timespantools` library provides an `AnnotatedTimespan` class which subclasses `Timespan` but adds a third read-only `annotation` property, which can be configured with any arbitrary object:

```python
>>> annotated_timespan = timespantools.AnnotatedTimespan(  ...
...     start_offset=(1, 8),
...     stop_offset=(7, 8),
...     annotation='Any arbitrary object can act as an annotation.'
... )
```

```python
>>> annotated_timespan.annotation
'Any arbitrary object can act as an annotation.'
```
Annotated timespans allow composers to position annotations or any other metadata along a timeline. If the annotation object is itself a mutable data structure such as a Python list or dictionary, that annotation can be used to store increasing amounts of information during the compositional process. Additionally, Abjad’s top-level `new()` function can be used to template new annotated timespans from old ones, replacing one annotation with another while preserving temporal information. If that same annotation object supports templating, nested reconfiguration can be performed:

```python
>>> metadata_timespan = new(annotated_timespan,
...    stop_offset=(3, 2),
...    annotation={
...        'durations': ((1, 8), (1, 8), (3, 16)),
...        'dynamic': indicatortools.Dynamic('ppp'),
...        'pitch_segment': pitchtools.PitchSegment([0, 1, 4, 7]),
...    },
...)
>>> metadata_timespan.annotation['bow_contact_point'] = Multiplier(1, 3)
>>> print(format(metadata_timespan))
timestools.AnnotatedTimespan(
    start_offset=durationtools.Offset(1, 8),
    stop_offset=durationtools.Offset(3, 2),
    annotation=
    {
        'bow_contact_point': durationtools.Multiplier(1, 3),
        'durations': (1, 8),
        'dynamic': indicatortools.Dynamic(name='ppp'),
        'pitch_segment': pitchtools.PitchSegment(
            pitchtools.NumberedPitch(0),
            pitchtools.NumberedPitch(1),
            pitchtools.NumberedPitch(4),
            pitchtools.NumberedPitch(7),
        ),
    }
)
```

Other `Timespan` subclasses are possible, allowing for even more configurable properties, as well as new methods. Two `Timespan` subclasses discussed later, `consert.PerformedTimespan` and `consert.SilentTimespan`, are core components in Consort’s score interpretation stage.
3.1.2 Time relations

Time relations model the disposition of one timespan relative to another timespan or offset. These relationships include intersection, containment, simultaneous start offsets or stop offsets, and many others. Abjad’s timespantools library provides a TimeRelation class and a collection of factory methods for configuring TimeRelation instances which formalize all possible dispositions of a timespan relative another timespan or offset\(^1\). Time relations may be configured with or without reference to any timespans or offsets at all, allowing for the possibility of modeling a purely abstract time relationship:

```python
>>> time_relation_1 = timespantools.timespan_2_intersects_timespan_1()
>>> print(format(time_relation_1))
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'),
     logical_operator='or',
    ), logical_operator='or'),
)
```

\(^1\)The thirty-three time relation factory functions contained in timespantools are `offset_happens_after_timespan_starts()`, `offset_happens_after_timespan_stops()`, `offset_happens_before_timespan_starts()`, `offset_happens_before_timespan_stops()`, `offset_happens_during_timespan()`, `offset_happens_when_timespan_starts()`, `offset_happens_when_timespan_stops()`, `timespan_2_contains_timespan_1_improperly()`, `timespan_2_curtails_timespan_1()`, `timespan_2_delays_timespan_1()`, `timespan_2_happens_during_timespan()`, `timespan_2_intersects_timespan()`, `timespan_2_is_congruent_to_timespan_1()`, `timespan_2_overlaps_all_of_timespan_1()`, `timespan_2_overlaps_only_start_of_timespan_1()`, `timespan_2_overlaps_only_stop_of_timespan_1()`, `timespan_2_overlaps_stop_of_timespan_1()`, `timespan_2_starts_after_timespan_1_starts()`, `timespan_2_starts_after_timespan_1_stops()`, `timespan_2_starts_after_timespan_1_start()`, `timespan_2_starts_after_timespan_1_stop()`, `timespan_2_starts_during_timespan()`, `timespan_2_starts_when_timespan_1_starts()`, `timespan_2_starts_when_timespan_1_stops()`, `timespan_2_stops_after_timespan_1_starts()`, `timespan_2_stops_after_timespan_1_stops()`, `timespan_2_stops_before_timespan_1_starts()`, `timespan_2_stops_before_timespan_1_stops()`, `timespan_2_stops_during_timespan()`, `timespan_2_stops_when_timespan_1_starts()`, `timespan_2_stops_when_timespan_1_stops()`, and `timespan_2_trisects_timespan_1()`.
The above intersection relationship between the timespans \([a, b)\) and \([c, d)\) can be described more pithily by the following predicate:

\[
(a \leq c \land c < b) \lor (c \leq a \land a < d)
\]  

(3.2)

A “half-configured” time relation is also possible. Such an object acts as a kind of “frozen” predicate. Calling the time relation as though it was a function\(^2\) on another timespan or offset returns a truth value:

```python
>>> time_relation_2 = timespantools.timespan_2_intersects_timespan_1(
...     timespan_1=timespantools.Timespan(0, 10),
...     
... )
>>> time_relation_2(timespan_2=timespantools.Timespan(5, 15))
True

>>> time_relation_2(timespan_2=timespantools.Timespan(30, 45))
False
```

Providing two timespans – or one timespan and an offset, as required – to one of the various time relation factory functions found in `timespantools` will configure and then immediately evaluate the generated time relation object, allowing the factory function itself to behave as a predicate:

```python
>>> timespantools.timespan_2_intersects_timespan_1(
...     timespan_1=timespantools.Timespan(1, 3),
...     timespan_2=timespantools.Timespan(2, 4),
... )
True
```

The time relation factory functions in `timespantools` are mirrored as methods on the `Timespan` class itself, allowing composers to determine the various relations of any timespan relative any other timespan or offset in an object-oriented fashion. The `Timespan` object automatically “fills in” the `timespan_1` argument to the `TimeRelation` with a reference to itself, and can pass the optional argument to its method call as the other object in the relation, allowing for the immediate evaluation of the relation as either true or false.

Consider the following three timespans:

```python
>>> timespan_1 = timespantools.Timespan(0, 10)
>>> timespan_2 = timespantools.Timespan(5, 15)
>>> timespan_3 = timespantools.Timespan(10, 20)
```

\(^2\) Any class in Python can be made treatable as a function by implementing a `__call__()` method.
After being collected into a *timespan inventory*, discussed further in section 3.2, the three timespans can be illustrated. Note that the beginning and end of each timespan is demarcated by a short vertical line, and that only non-intersecting timespans are shown in each “row” in the illustration:

![Diagram of three timespans]

We can test for intersection between these three timespans via the `intersects_timespan()` method. Two timespans are considered to intersect if any part of one timespan overlaps any part of another. *Intersection* is therefore commutative. Note that `timespan_1` and `timespan_3` do not overlap even though they share the offset 10. As discussed earlier, timespans are left-closed / right-open, meaning that while their start offset is contained in the infinite set of offsets they range over, their stop offset is not:

```python
>>> timespan_1.intersects_timespan(timespan_2)
True
>>> timespan_1.intersects_timespan(timespan_3)
False
>>> timespan_2.intersects_timespan(timespan_1)
True
>>> timespan_2.intersects_timespan(timespan_3)
True
>>> timespan_3.intersects_timespan(timespan_1)
False
>>> timespan_3.intersects_timespan(timespan_2)
True
```

*Congruency* tests whether two timespans share the same start and stop offset. Every timespan is congruent with itself:

```python
>>> timespan_1.is_congruent_to_timespan(timespan_2)
False
>>> timespan_1.is_congruent_to_timespan(timespan_1)
True
```

*Tangency* tests whether one timespan’s stop offset is the same as another timespan’s start offset, or vice versa. Tangency can be used to determine if a sorted collection of timespans is both entirely contiguous and non-overlapping:

```python
>>> timespan_1.is_tangent_to_timespan(timespan_2)
False
```
A wide variety of other time relation predicates are also possible, such as testing if a timespan intersects with a specific offset, testing if a timespan overlaps only the beginning or end of another timespan, or testing if a timespan contains another timespan entirely. These predicates make possible the many generative operations carried out on timespans.

3.1.3 Operations on timespans

Many Timespan methods provide transformations, such as translation, scaling or offset rounding. Because timespans are immutable, these methods create a new timespan based on the old one and then return the new, leaving the old exactly as it was:

```python
>>> timespan = timespanTools.Timespan(0, 15)
>>> timespan.translate(3)
Timespan(start_offset=Offset(3, 1), stop_offset=Offset(18, 1))

>>> timespan.scale(3)
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(45, 1))

>>> timespan.round_offsets(2)
Timespan(start_offset=Offset(0, 1), stop_offset=Offset(16, 1))
```

These generative methods are implemented internally in terms of templating via `new()`, as described in section 3.1, allowing annotated timespan subclasses to partake in the same generative functionality – reconfiguring their start and / or stop offsets, but maintaining all other previously configured properties:

```python
>>> annotated_timespan = timespanTools.AnnotatedTimespan(0, 5, 'an annotation')
>>> scaledannotated_timespan = annotated_timespan.translate((-1, 3))

>>> print(format(scaledannotated_timespan))
timespanTools.AnnotatedTimespan(
    start_offset=durationTools.Offset(-1, 3),
    stop_offset=durationTools.Offset(14, 3),
    annotation='an annotation',
)
```

Some generative operations may return zero or more timespans, aggregated in a data structure called a **timespan inventory**, discussed at length in section 3.2. Splitting a timespan by an offset is one such operation. If the timespan to
be split properly contains the splitting offset, a timespan inventory containing two new timespans will be returned. Otherwise, the split operation will return a timespan inventory containing a copy of the original input timespan:

```python
>>> two_shards = timespan.split_at_offset(5)
>>> print(format(two_shards))
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(15, 1),
    ),
    ]
)
```

```python
>>> one_shard = timespan.split_at_offset(10000)
>>> print(format(one_shard))
timespantools.TimespanInventory(
    [timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(15, 1),
    ),
    ]
)
```

More complex operations between timespans resulting in timespan inventories include subtraction and the logical operations AND, OR and XOR. These generative operations are conceptually set operations – union, difference, intersection, symmetric difference, etc. – performed on the two sets of offsets contained by the two timespan operands. Consider these same set operations – union, difference, intersection, symmetric difference – carried out on trivial sets in Python:

```python
>>> set([1, 2, 3]) | set([2, 3, 4])  # union
set([1, 2, 3, 4])
```

```python
>>> set([1, 2, 3]) - set([2, 3, 4])  # difference
set([1])
```

```python
>>> set([1, 2, 3]) & set([2, 3, 4])  # intersection
set([2, 3])
```

3 Proper containment of an offset means that the offset is greater than the timespan’s start offset and less than the timespan’s stop offset. Improper containment would indicated the offset is greater than or equal to the timespan’s start offset and less than or equal to its stop offset.
Set operations performed on timespans are conceptually identical, but operate on infinite but bounded sets of offsets instead of discrete sets of objects such as integers. For example, subtracting one timespan from another computes the set difference of the offsets contained by both. This operation is not commutative – subtracting one timespan from another will not result in the same output as subtracting the latter from the former. Subtracting a timespan from itself always results in the empty set of offsets: no timespan at all:

```python
>>> result = timespantools.Timespan(0, 10) - timespantools.Timespan(0, 10)
>>> print(format(result))
timespantools.TimespanInventory(
    [],
)
```

```python
>>> result = timespantools.Timespan(0, 10) - timespantools.Timespan(5, 15)
>>> print(format(result))
timespantools.TimespanInventory(
    [
        timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(5, 1),
        ),
    ],
)
```

```python
>>> result = timespantools.Timespan(0, 10) - timespantools.Timespan(10, 20)
>>> print(format(result))
timespantools.TimespanInventory(
    [
        timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(10, 1),
        ),
    ],
)
```

```python
>>> result = timespantools.Timespan(5, 15) - timespantools.Timespan(0, 10)
>>> print(format(result))
timespantools.TimespanInventory(
    [
        timespantools.Timespan(
            start_offset=durationtools.Offset(10, 1),
            stop_offset=durationtools.Offset(15, 1),
        ),
    ],
)
```

```python
>>> result = timespantools.Timespan(5, 15) - timespantools.Timespan(5, 15)
>>> print(format(result))
87
```
Computing the logical OR of two timespans results in an offset set union – a commutative operation, effectively fusing the timespans together if they overlap:
stop_offset=durationtools.Offset(15, 1),
],
)

>>> result = timespantools.Timespan(0, 10) | timespantools.Timespan(10, 20)
>>> print(format(result))
timespantools.TimespanInventory(
    [timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(20, 1),
    ),
    ]
)

>>> result = timespantools.Timespan(5, 15) | timespantools.Timespan(10, 20)
>>> print(format(result))
timespantools.TimespanInventory(
    [timespantools.Timespan(
        start_offset=durationtools.Offset(5, 1),
        stop_offset=durationtools.Offset(20, 1),
    ],
    ]
)

Unioning guarantees that all of the offsets contained in the two input timespans will appear in the output timespan or timespans, whether or not any overlap occurred:

>>> result = timespantools.Timespan(10, 20) | timespantools.Timespan(25, 50)
>>> print(format(result))
timespantools.TimespanInventory(
    [timespantools.Timespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1),
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(25, 1),
        stop_offset=durationtools.Offset(50, 1),
    ),
    ]
)

The logical **AND** – set intersection – results in the intersection of the two input timespans. Only those offsets which occur in both timespan operands will occur in the output timespan, if any:

>>> result = timespantools.Timespan(0, 10) & timespantools.Timespan(5, 15)
>>> print(format(result))
Logical XOR – also known as exclusive OR – results in the symmetric difference of the two input timespans. Only those offsets which are contained by only one of the two input timespans will occur in the output:

```python
>>> result = timespantools.Timespan(0, 10) ^ timespantools.Timespan(0, 15)
>>> 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(10, 1),
        stop_offset=durationtools.Offset(15, 1),
    ),
])
>>> result = timespantools.Timespan(10, 20) ^ timespantools.Timespan(0, 15)
>>> 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(10, 1),
        stop_offset=durationtools.Offset(15, 1),
    ),
])
>>> result = timespantools.Timespan(0, 10) ^ timespantools.Timespan(10, 20)
>>> print(format(result))
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1),
    )
])
```
These operations, while perhaps initially rather abstract, are incredibly powerful and artistically useful. They afford composers with the procedural building blocks to mask temporal objects with one another, fuse them together, or create lacunae. When extended to operate on many timespans at once, wholesale transformations on massed timespans becomes practical.

### 3.2 Timespan inventories

Abjad provides a `TimespanInventory` class specifically for aggregating together a collection of `Timespan` objects. Timespan inventories implement Python’s *mutable sequence protocol* which allow them to behave exactly like lists, supporting appension, extension, insertion, indexing, sorting, iteration and other procedures pertinent to list-like objects. They also provide a wide variety of properties and methods for interacting with massed groups of timespans such as searching for timespans matching a time relation, splitting all timespans which intersect with a given offset, mapping a logical operation against all timespans in the inventory, or partitioning an inventory containing overlapping timespans into multiple separate inventories containing non-overlapping timespans.

---

4 Consort provides its own timespan collection class – the `TimespanCollection`. This class stores timespans internally not in a list, but in a balanced *interval tree*. An interval tree is an augmented, self-balancing binary tree which stores start offsets and stop offsets. Such a data structure guarantees its contents are always sorted, and allows for highly optimized lookups for timespan matching various search criteria. The `TimespanCollection` class is used at crucial points during Consort’s interpretation stage simply for purposes of speed, and should be considered an implementation detail. It provides only a few methods, specifically for affording rapid search and retrieval of timespans intersecting other timespans or offsets. With work, its internal data structure may eventually be merged into Abjad’s own `TimespanInventory` class.
Like a Python list, a timespan inventory can be created with an iterable of zero or more timespans as an instantiation argument, be appended to, or extended into:

```python
>>> timespan_inventory = timespantools.TimespanInventory([...
     ...  timespantools.Timespan(0, 16),
     ...  ])

>>> timespan_inventory.append(timespantools.Timespan(5, 12))

>>> timespan_inventory.extend([...
     ...  timespantools.Timespan(-2, 8),
     ...  timespantools.Timespan(15, 20),
     ...  ])

>>> print(format(timespan_inventory))
TimespanInventory([
    Timespan(start_offset=durationtools.Offset(0, 1),
             stop_offset=durationtools.Offset(16, 1)),
    Timespan(start_offset=durationtools.Offset(5, 1),
             stop_offset=durationtools.Offset(12, 1)),
    Timespan(start_offset=durationtools.Offset(-2, 1),
             stop_offset=durationtools.Offset(8, 1)),
    Timespan(start_offset=durationtools.Offset(15, 1),
             stop_offset=durationtools.Offset(20, 1)),
]
```

Timespan inventories can also be queried for their length, be indexed into, or iterating over like any other sequence-like object in Python:

```python
>>> len(timespan_inventory)
4

>>> timespan_inventory[1]
Timespan(start_offset=Offset(5, 1), stop_offset=Offset(12, 1))

>>> for timespan in timespan_inventory:
    ...
    timespan
```

-2
1
0
1
5
1
8
1
12
1
15
1
16
1
20
1

Timespan inventories can also be queried for their length, be indexed into, or iterated over like any other sequence-like object in Python:
One timespan inventory can be created from another by passing the first timespan inventory as an instantiation argument to the second, just like one can create a Python list simply by calling list() on another iterable object, including another list. Unlike timespans, timespan inventories are mutable. As many of the operations implemented on timespan inventories mutate the inventory in-place, this instantiation pattern provides a simple means of “copying”, allowing composers to duplicate a timespan structure before operating on it, thereby preserving the original:

```python
>>> duplicate = timespantools.TimespanInventory(timespan_inventory)
>>> duplicate == timespan_inventory
True
>>> duplicate is timespan_inventory
False
>>> print(format(duplicate))
timespantools.TimespanInventory(
    [
        timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(16, 1),
        ),
        timespantools.Timespan(
            start_offset=durationtools.Offset(5, 1),
            stop_offset=durationtools.Offset(12, 1),
        ),
        timespantools.Timespan(
            start_offset=durationtools.Offset(-2, 1),
            stop_offset=durationtools.Offset(8, 1),
        ),
        timespantools.Timespan(
            start_offset=durationtools.Offset(15, 1),
            stop_offset=durationtools.Offset(20, 1),
        ),
    ]
)
```

Timespan inventories can be treated as timespans themselves, having a start offset equal to the minimum start offset of any of their contained timespans, and a stop offset equal to the maximum stop offset of any of their contained timespans. Their start and stop offset properties allow them to express a duration, as well as provide a concrete timespan representation. Because timespan inventories can be modeled as timespans, they can even be inserted into other timespan inventories, effectively masquerading as timespans:
Because timespan inventories aggregate multiple timespans together, they also provide properties for describing collective qualities of those timespans. **Contiguity** tests if every timespan in the inventory is tangent to another timespan and also does not overlap any other timespan. **Overlap** tests if any timespan intersects any other timespan. **Well-formedness** tests that all timespans’ durations are greater than 0. A timespan inventory whose timespans are contiguous is necessarily also non-overlapping:

```
>>> timespan_inventory.all_are_contiguous
False

>>> timespan_inventory.all_are_nonoverlapping
False

>>> timespan_inventory.all_are_well_formed
True
```

The following timespan inventory’s timespans are non-overlapping but also non-contiguous:

```
>>> timespan_inventory = timespantools.TimespanInventory([
    ...
    timespantools.Timespan(0, 10),
    ...
    timespantools.Timespan(10, 20),
    ...
    timespantools.Timespan(30, 40),
    ...
])
>>> show(timespan_inventory)

0                     10                     20                     30                     40

```

```
>>> timespan_inventory.all_are_contiguous
False

>>> timespan_inventory.all_are_nonoverlapping
True
```

In contrast, this timespan inventory’s timespans are both non-overlapping and contiguous:
>>> timespan_inventory = timespantools.TimespanInventory([
...    timespantools.Timespan(0, 16),
...    timespantools.Timespan(5, 12),
...    timespantools.Timespan(-2, 8),
...  ])

>>> show(timespan_inventory)

>>> timespan_inventory.all_are_contiguous
True

>>> timespan_inventory.all_are_nonoverlapping
True

### 3.2.1 Operations on Timespan Inventories

Timespan inventories implement unioning, differencing and splitting methods which parallel those implemented on timespans themselves. These methods map the desired operation onto the contents of the inventory by, for example, splitting every timespan contained in a given inventory by some offset. All of these operations act in place. The intersection of all of the timespans in a timespan inventory relative another timespan can be computed with the Python & operator, the same syntax used when operating on individual timespans, as demonstrated in subsection 3.1.3: 

```python
>>> timespan_inventory = timespantools.TimespanInventory([
...    timespantools.Timespan(0, 16),
...    timespantools.Timespan(5, 12),
...    timespantools.Timespan(-2, 8),
...  ])

>>> show(timespan_inventory)

>>> timespan_operand = timespantools.Timespan(6, 10)

>>> result = timespan_inventory & timespan_operand

>>> show(result, range_=(-2, 16))
```

The call to `show()` in the code example here contains the keyword argument `range_`. This argument controls the horizontal scaling and spacing of the timespan illustration, allowing different illustrations to be aligned against one another in a document even if the timespan inventories they illustrate have different start and stop offsets. The name `range_` – with a trailing underscore – is used instead of `range`, a commonly-used built-in Python function. It is common practice in the Python community to append underscores to names when they would otherwise conflict with reserved words in Python’s grammar or global built-in namespace. For example, one would use the names `break_`, `object_`, `set_` and `type_` instead of the reserved word `break`, or the built-in names `object`, `set` and `type`. 

---

5The call to `show()` in the code example here contains the keyword argument `range_`. This argument controls the horizontal scaling and spacing of the timespan illustration, allowing different illustrations to be aligned against one another in a document even if the timespan inventories they illustrate have different start and stop offsets. The name `range_` – with a trailing underscore – is used instead of `range`, a commonly-used built-in Python function. It is common practice in the Python community to append underscores to names when they would otherwise conflict with reserved words in Python’s grammar or global built-in namespace. For example, one would use the names `break_`, `object_`, `set_` and `type_` instead of the reserved word `break`, or the built-in names `object`, `set` and `type`. 

---

95
Likewise, the offsets bound by a given timespan can be subtracted from all of the timespans in a timespan inventory, effectively cutting a hole in that inventory’s timeline:

```python
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 16),
...     timespantools.Timespan(5, 12),
...     timespantools.Timespan(-2, 8),
... ])
>>> show(timespan_inventory)
```

```text
-2 1 0 1 5 1 8 1 12 1 16
```

```python
>>> timespan_operand = timespantools.Timespan(6, 10)
```

```python
>>> result = timespan_inventory - timespan_operand
>>> show(result)
```

```text
-2 1 0 1 5 1 6 1 10
```

As with a single timespan and an offset, an entire timespan inventory can be split into two separate inventories via the `split_at_offset()` method:

```python
>>> timespan_inventory = timespantools.TimespanInventory([
...     timespantools.Timespan(0, 3),
...     timespantools.Timespan(3, 6),
...     timespantools.Timespan(6, 10),
... ])
>>> show(timespan_inventory)
```

```text
0 1 3 1 6 1 10
```

```python
>>> left, right = timespan_inventory.split_at_offset(4)
>>> show(left, range_=(0, 10))
```

```text
0 1 3 1 4
```

```python
>>> show(right, range_=(0, 10))
```

```text
4 1 6 1 10
```

The `TimespanInventory` class also provides the convenience method `split_at_offsets()` for splitting an inventory by an arbitrary number of offsets at once:
The mutating methods described above modify a timespan inventory by mapping some procedure against its contents and some outside timespan or offset. However, timespan inventories may also be modified by applying a procedure solely against the contents of the inventory itself, mapping each timespan in the collection against each other timespan in that collection. For example, a timespan inventory can be modified by computing the logical OR – the set union – of every timespan in the inventory relative every other timespan, effectively fusing all overlapping timespans together:
A timespan inventory can also be modified by computing the logical AND of every timespan in the inventory relative to every other timespan. This procedure leaves only those offsets where every single timespan overlaps:

```python
>>> timespan_inventory = timespantools.TimespanInventory([
    ...    timespantools.Timespan(-2, 8),
    ...    timespantools.Timespan(0, 10),
    ...    timespantools.Timespan(5, 12),
    ... ])
>>> show(timespan_inventory)
```

```plaintext
-2
1
12
1
```

```python
>>> result = timespan_inventory.compute_logical_and()
>>> print(format(result))
timespantools.TimespanInventory(
    [
        timespantools.Timespan(
            start_offset=durationtools.Offset(5, 1),
            stop_offset=durationtools.Offset(8, 1),
            )
    ])
```

```plaintext
-2
1
0
1
```

Lastly, computing the in-place logical XOR removes all overlap from the timespan inventory, leaving only those offsets occupied by only one timespan:

```python
>>> timespan_inventory = timespantools.TimespanInventory([
    ...    timespantools.Timespan(-2, 8),
    ...    timespantools.Timespan(0, 10),
    ...    timespantools.Timespan(5, 12),
    ... ])
>>> show(timespan_inventory)
```

```plaintext
-2
1
2
1
```

```python
>>> result = timespan_inventory.compute_logical_xor()
>>> print(format(result))
timespantools.TimespanInventory(
    [
    ]
)
Timespan partitioning separates a timespan inventory into groups of overlapping and optionally tangent timespans, aggregated into new timespan inventories. This procedure allows composers to isolate contiguous blocks of activity:

```python
>>> show(result, range_=(-2, 12))
```

```python
>>> timespan_inventory = timespantools.TimespanInventory([
...    timespantools.Timespan(0, 10),
...    timespantools.Timespan(5, 15),
...    timespantools.Timespan(15, 20),
...    timespantools.Timespan(25, 30),
...])
>>> show(timespan_inventory)
```

```python
>>> for shard in timespan_inventory.partition():
...    show(shard, range_=(0, 30))
```

```python
>>> for shard in timespan_inventory.partition(include_tangent_timespans=True):
...    show(shard, range_=(0, 30))
```
Conversely, *explosion* separates a timespan inventory into one or more new inventories in an attempt to limit the amount of overlap in each resulting inventory. The number of output inventories can be left unspecified, in which case explosion will generate as many inventories as necessary to prevent overlap entirely in every resulting inventory:

```python
>>> timespan_inventory = timespan_tools.TimespanInventory([
...     timespan_tools.Timespan(-2, 8),
...     timespan_tools.Timespan(-2, 1),
...     timespan_tools.Timespan(0, 16),
...     timespan_tools.Timespan(4, 7),
...     timespan_tools.Timespan(4, 11),
...     timespan_tools.Timespan(5, 12),
...     timespan_tools.Timespan(11, 13),
...     timespan_tools.Timespan(14, 17),
...     timespan_tools.Timespan(15, 20),
... ])
>>> show(timespan_inventory)
```

The number of output inventories can also be set explicitly, in which case explosion will attempt to limit the amount of overlap as much as possible, while maintaining a similar level of density across each resulting inventory:

```python
>>> for shard in timespan_inventory.explode(inventory_count=2):
...     show(shard, range_=(-2, 20))
```

---

*Explosion* is one of the techniques used to create timespan inventory illustrations, providing the mechanism by which arbitrary collections of timespans are automatically separated into non-overlapping inventories, which can then be rendered graphically as “rows” of line segments.
The procedures outlined above provide high-level tools for interacting with large numbers of timespans at once. All of the techniques described in chapter 4 with regards to Consort’s score interpretation stage – timespan consolidation, cascading overlap resolution, multiplexing multiple inventories into one, demultiplexing one inventory into many, and so forth – build on and extend these techniques.

3.3 Annotated timespans in Consort

While timespans and timespan inventories provide a very general model for modeling the disposition of durated events in time, a larger question remains: how can composers create enough timespans, and in various patterns, to be musically interesting? Consort approaches this problem by providing a collection of factory classes – timespan-makers – which can be configured and called to create arbitrarily large amounts of timespans. However, before turning to a detailed discussion of timespan-makers, we must first discuss the products of the timespan themselves.

Consort provides two separate timespan subclasses which are integral, if transient, components of its score interpretation stage: the PerformedTimespan and SilentTimespan classes. These classes are never created “by hand” during Consort’s specification stage – this is, explicitly instantiated by a composer while specifying a score segment –, but are instead generated as transitory objects during interpretation. Consort uses PerformedTimespan objects to indicate locations in the score timeline where some active musical material should appear, while SilentTimespan objects indicate tacet passages.

As will be described in more detail in chapter 4, Consort requires composers to specify musical materials in layers, and to specify specifically in which voice contexts in the score – vertically – that material should occur. During the course of interpretation, Consort organizes generated timespans by voice name and layer into separate timespan inventories, and then progressively masks out timespans in timespan inventories with lower layer numbers by those timespans with higher layer numbers. One can imagine this process as analogous to the use of opaque overlapping layers in image editing software. Both the PerformedTimespan and SilentTimespan classes provide configurable properties for layer and voice name, in addition to the start offset and stop offset properties provided by their parent Timespan class. These properties allow the processes that generate them to record when they were created, as well as where the should appear in the score, should they survive the masking process:

```python
>>> performed_timespan = consort.PerformedTimespan(  
    ...     layer=1,  
    ...     start_offset=(1, 2),  
    ...     stop_offset=(7, 8),
```
Performed timespans possess a number of other configurable properties used during score interpretation. These include `minimum_duration` – used to force erasure of timespans considered too short, often because they have been partially masked, `music_specifier` – used to attach information about how to generate notation from a timespan, `divisions` – a memento of the original duration structure of a contiguous sequence of performed timespans which have been consolidated into a single performed timespan after the masking process ends, and `music` – used to attach any generated notation prior to maquetting:

```python
>>> performed_timespan = consort.PerformedTimespan(
...     layer=1,
...     minimum_duration=Duration(1, 8),
...     music_specifier=consort.MusicSpecifier(),
...     start_offset=Offset(1, 4),
...     stop_offset=Offset(2, 1),
...     voice_name='Violin 1 LH Voice',
... )
```

Unlike performed timespans, silent timespans have no explicit reality in score, but can still be created for a particular voice context and layer. They act to simply erase any timespan in a lower layer through masking. This allows timespan-making processes to demand not only when a musical event should happen, but also when musical events should not. For example, silent timespans make it possible to demand silence before music events involving instrument or mallet changes:

```python
>>> silent_timespan = consort.SilentTimespan(
...     layer=2,
...     start_offset=Offset(0, 1),
...     stop_offset=Offset(1, 4),
...     voice_name='Oboe Voice',
... )
```

All of the above properties are explained in more depth in chapter 4, in context of Consort’s interpretation process. For the sake of demonstrating timespan-creation principles pithily, the `music_specifier` property of any performed timespan can be set to an arbitrary object such as Python’s `None` rather than a `MusicSpecifier` instance as actually done in practice.
3.4 **Timespan-makers**

Consort provides a family of factory classes for producing timespans, each implementing a different strategy for populating timespan inventories, but all unified via the same callable interface. Timespan-makers take as input a mapping of voice-names to *music specifiers* – arbitrary objects specifying how a given timespan might be rendered as notation –, a *target* timespan indicating the minimum and maximum permitted start offsets of any timespan created by the timespan-maker, an optional timespan inventory to modify in-place, and an optional *layer* identifier indicating in which pass a particular timespan was created.\(^7\) All timespan-makers produce timespan inventories as output.

Timespan-makers fall into two broad categories: independent and dependent. Independent timespan-makers create timespans without regard for any pre-existing timespans, and therefore do not require being called with a timespan inventory instance. Dependent timespan-makers create their timespans based on the contents of a pre-existing timespan inventory, basing their output on various aspects of the structure of their input.

Additionally, all timespan-makers can be configured with *padding* and *timespan specifier* keywords. Padding allows the timespan-maker to force “silence” – in the form of *SilentTimespan* objects – around the beginnings and ends of contiguous groups of timespans it creates. Timespan specifiers provide templates for some of the configurable properties specific to performed timespans, such as *minimum_duration* and *forbid_splitting*, allowing the timespan-maker to apply the configuration defined in the timespan specifier to each of the timespans it creates.

### 3.4.1 Flooded timespan-makers

*Flooded* timespan-makers implement the most trivial timespan-creation strategy in the timespan-maker class family. Their name derives from the ubiquitous graphic-design “flood fill” tool, which fills an entire connected area with the same color or texture. Flooded timespan-makers create one timespan for each voice in the input voice-name-to-music-specifier mapping, filling the entirety of the provided target timespan from beginning to end.

For example, the following flooded timespan-maker will create a timespan inventory populated by a single performed timespan associated with a “Violin Voice” context, completely filling the span of its target timespan, from \(\frac{1}{4}\) to \(\frac{11}{8}\):

---

\(^7\)Layer identifiers allow timespans to be sorted not only by their position in time or by their associated voice name, but also by at which point – which *layer* – during some compositional process they were created. Layer ordering allows masking processes to mask “earlier” timespans with “later” ones.
```python
>>> music_specifiers = {'Violin Voice': 'violin music'}
>>> target_timespan = timespantools.Timespan((1, 4), (11, 8))
>>> flooded_timespan_maker = consort.FloodedTimespanMaker()
>>> timespan_inventory = flooded_timespan_maker(  
...     music_specifiers=music_specifiers,  
...     target_timespan=target_timespan,  
... )
>>> print(format(timespan_inventory))
timespantools.TimespanInventory(
    [  
        consort.tools.PerformedTimespan(  
            start_offset=durationtools.Offset(1, 4),  
            stop_offset=durationtools.Offset(11, 8),  
            music_specifier='violin music',  
            voice_name='Violin Voice',  
        ),
    ]
)
```

Adding a second entry to the music specifier mapping results in two timespans in the output. Likewise, calling the timespan-maker with a `layer` keyword configures the output timespans with that layer number. The layer is indicated in the illustration just above and to the right of the beginning of each timespan:

```python
>>> music_specifiers = {
...     'Violin Voice': 'violin music',
...     'Cello Voice': 'cello music',
... }
>>> timespan_inventory = flooded_timespan_maker(  
...     layer=3,
...     music_specifiers=music_specifiers,  
...     target_timespan=target_timespan,  
... )
>>> print(format(timespan_inventory))
timespantools.TimespanInventory(
    [  
        consort.tools.PerformedTimespan(  
            start_offset=durationtools.Offset(1, 4),  
            stop_offset=durationtools.Offset(11, 8),  
            layer=3,  
            music_specifier='cello music',  
            voice_name='Cello Voice',  
        ),  
        consort.tools.PerformedTimespan(  
            start_offset=durationtools.Offset(1, 4),  
            stop_offset=durationtools.Offset(11, 8),  
            layer=3,  
            music_specifier='violin music',  
            voice_name='Violin Voice',  
        ),
    ]
)
```
Configuring the timespan-maker with padding creates silent timespans around the beginning and end of each group of output timespans on a per-voice basis. The timespan-maker will also configure these silent timespans with any specified layer number, allowing them to be sorted along with performed timespans from the same timespan-creation pass. Silent timespans are illustrated with dashed line to distinguish them from performed timespans:
minimum_duration=durationtools.Duration(1, 8),
    music_specifier='violin music',
    voice_name='Violin Voice',
),
    consort.tools.SilentTimespan(
        start_offset=durationtools.Offset(11, 8),
        stop_offset=durationtools.Offset(13, 8),
        layer=5,
        voice_name='Violin Voice',
    ),
    consort.tools.SilentTimespan(
        start_offset=durationtools.Offset(11, 8),
        stop_offset=durationtools.Offset(13, 8),
        layer=5,
        voice_name='Cello Voice',
    ),
]}

>>> show(timespan_inventory, key='voice_name')

Cello Voice:

\begin{tabular}{c}
| 0 | 1 | 1 | 4 | 11 | 8 | 13 | 8 | 5 | 5 | 5 |
\hline
\end{tabular}

Violin Voice:

\begin{tabular}{c}
| 0 | 1 | 1 | 4 | 11 | 8 | 13 | 8 | 5 | 5 | 5 |
\hline
\end{tabular}

3.4.2 Talea timespan-makers

Consort’s TaleaTimespanMaker class creates rich timespan textures through the use of talea – infinitely cyclic duration patterns. Abjad implements the concept of talea via the Talea class in its rhythmmakertools library, which is discussed at length in section 3.5. Talea objects define their infinitely cyclic sequence of durations in terms of a finite sequence of numerators paired with a single denominator. Once defined, they can be iterated over and indexed like an infinite sequence, simplifying the process of generating looped patterns of durations of arbitrary lengths:

```python
>>> talea = rhythmmakertools.Talea(
    ...    counts=[1, 2, 3, 4],
    ...    denominator=16,
    ...)
>>> for index in range(10):
    ...    talea[index]
    ...
```

... NonreducedFraction(1, 16)
   NonreducedFraction(2, 16)
   NonreducedFraction(3, 16)
   NonreducedFraction(4, 16)
   NonreducedFraction(1, 16)
   NonreducedFraction(2, 16)
   NonreducedFraction(3, 16)
Talea timespan-makers make use of talea to control patterns of performed durations and silences. These timespan-makers can create timespans in *synchronized* or *unsynchronized* fashions. Unsynchronized timespan generation proceeds voice-by-voice through the input voice-name-to-music-specifier mapping, creating timespans from the start of the target timespan until its stop, then wrapping around to the next voice. Such unsynchronized timespan creation is ideal for creating massed textures of seemingly-unrelated timespans. Careful management of silence patterns and contiguous timespan group lengths during unsynchronized creation can create dense, overlapping textures or sparse, pointillistic ones. Contrastingly, synchronized creation provides a mechanism for creating moments of shared attack across voices, followed by shared silences:

```python
>>> talea_timespan_maker = consort.TaleaTimespanMaker()
>>> print(format(talea_timespan_maker))
consort.tools.TaleaTimespanMaker(
    playing_talea=rhythmmakertools.Talea(
        counts=(4,),
        denominator=16,
    ),
    playing_groupings=(1,),
    repeat=True,
    silence_talea=rhythmmakertools.Talea(
        counts=(4,),
        denominator=16,
    ),
    step_anchor=Right,
    synchronize_groupings=False,
    synchronize_step=False,
)
```

All of the timespan inventories created in the following talea timespan-maker examples make use of the same music specifier mapping and target timespan. An ordered dictionary from Python’s `collections` module guarantees that the timespan-makers process the music specifier entries in the same order, from “Voice 1” through “Voice 4”:

---

8The hash implementation used by Python’s dictionary class does not guarantee any particular ordering of keys, and may differ from one version of Python to the next, or even from one machine architecture to the next. While the timespan inventory illustrations shown with voice-name labels appear ordered, that is because of explicit lexical sorting of the voice-names in the
Without any manual configuration, talea timespan-makers generate unsynchronized 1-length groups of $\frac{1}{4}$-duration timespans, separated by $\frac{1}{4}$-duration silences, creating the appearance of synchronization across voices:

Changing the talea timespan-maker’s `playing_talea` property from a series of $\frac{1}{4}$-durations to $\frac{3}{4}, \frac{2}{4}, \frac{1}{4}$ reveals the timespan-maker’s voice-wrapping behavior. Note how “Voice 1”的 timespans receive the durations $\frac{1}{4}, \frac{3}{4}, \frac{2}{4}, \frac{1}{4}$ and $\frac{3}{4}$. “Voice 2” continues the duration sequence with $\frac{3}{4}, \frac{2}{4}, \frac{1}{4}$ and so forth. The playing-duration talea continues to wrap around the end of each voice’s timespans to the beginning of the next voice’s:

illustration algorithm, not because of any particular ordering in the timespan-maker’s voice-name-to-music-specifier mapping. Timespan-makers iterate over the keys in that mapping when creating timespans, and plain dictionaries may lead to unexpected results. For that reason, these examples – and Consort itself – make use of the `OrderedDict` class from Python’s `collections` module, which guarantees that keys be ordered by when they were inserted.

108
The talea timespan-maker’s `playing_groupings` property controls how many timespans are created together as contiguous groups. When a grouping would run beyond the stop offset provided by the target timespan, that grouping’s length is chosen again for the first timespan group of the next voice’s timespans. Note how “Voice 2”’s final timespan, from $\frac{13}{4}$ to $\frac{3}{4}$, should—according to the following talea timespan-maker’s groupings pattern—begin a group of length 2. Because that grouping would run beyond the target timespan’s $\frac{19}{4}$ stop offset, the following voice’s first timespan group receives the 2-length grouping instead:

Reconfiguring the above talea timespan-maker with a different `silence_talea` produces patterned variations in the durations of silences between timespan groups:
...  silence_talea=rhythmmakertools.Talea(
...    counts=(3, 1, 1),
...    denominator=8,
...  ),
...)
>>> result = talea_timespan_maker(
...    music_specifiers=music_specifiers,
...    target_timespan=target_timespan,
...  )
>>> show(result, key='voice_name')

Voice 1:

Voice 2:

Voice 3:

Voice 4:

Likewise, changing the initial_silence_talea produces varying amounts of silence at the beginning of each voice:

>>> talea_timespan_maker = new(
...    talea_timespan_maker,
...    initial_silence_talea=rhythmmakertools.Talea(
...      counts=(0, 1, 3),
...      denominator=8,
...    ),
...)
>>> result = talea_timespan_maker(
...    music_specifiers=music_specifiers,
...    target_timespan=target_timespan,
...  )
>>> show(result, key='voice_name')

Voice 1:

Voice 2:

Voice 3:

Voice 4:

Talea timespan-makers also provide for transformations derived from the timespan inventory class itself. The reflect keyword configures the timespan-maker to reflect its output timespan inventory around its own axis, creating a “mirror image”:

110
Synchronized talea timespan-makers can be created simply by configuring a new timespan-maker with its synchronize_step flag set to true. This option causes the timespan-maker to create all of the timespans in every voice at once, then select some amount of silence between the end of that vertically-synchronized group and the beginning of the next. Note here how the duration of each timespan follows the timespan-maker’s \(\frac{1}{4}, \frac{2}{4}, \frac{3}{4}, \frac{4}{4}, \frac{5}{4}\) pattern not from left-to-right by voice, but top-to-bottom by voice and then left-to-right:
The silence durations between these synchronized groups are calculated from the end of the longest timespan in each group to the beginning of the next group. Therefore the second group starts at $\frac{1}{4}$ because the first silence duration is $\frac{4}{8}$ and the first group stops at $\frac{1}{2}$. Similarly, the third group starts at $\frac{5}{2} (\frac{20}{8})$ because the second silence duration is $\frac{7}{8}$ and the second group ended at $\frac{13}{8}$. This silence stepping can also be calculated from the beginning of one group to the next, rather than from the end of one to the beginning of the next, by changing the timespan-maker’s step_anchor property from Right to Left. Such a change helps guarantee the timing of initial attacks across synchronized groups.

Initial silences behave differently in synchronized talea timespan-makers than with unsynchronized talea timespan-makers. Rather than adding silences only at the very beginning of each voice’s timespans, silences are determined for each voice at the beginning of each synchronized group. Note how “Voice 3” is missing a fourth timespan. That timespan, with a duration of $\frac{5}{8}$, and an initial silence of $\frac{2}{8}$ measured from the offset $\frac{4}{1}$, would have extended to $\frac{39}{8} - \frac{1}{8}$ more than the target timespan’s $\frac{19}{4}$ stop offset:

```python
>>> synchronized_talea_timespan_maker = new(
    ...   synchronized_talea_timespan_maker,
    ...   initial_silence_talea=rhythmmakertools.Talea(
    ...     counts=(0, 1, 2),
    ...     denominator=8,
    ...   ),
    ... )
>>> result = synchronized_talea_timespan_maker(
    ...   music_specifiers=music_specifiers,
    ...   target_timespan=target_timespan,
    ... )
>>> show(result, key='voice_name')
```
Unlike flooded timespan-makers, padding durations are included in the start offset and step duration calculations for talea timespan-makers:

```python
>>> synchronized_talea_timespan_maker = new(  
...     synchronized_talea_timespan_maker,  
...     padding=(1, 8),  
... )
>>> result = synchronized_talea_timespan_maker(  
...     music_specifiers=music_specifiers,  
...     target_timespan=target_timespan,  
... )
>>> show(result, key='voice_name')
```

3.4.3 Dependent timespan-makers

*Dependent* timespan-makers create timespans based on the disposition of timespans in an input timespan inventory, selecting the start and stop offsets of their generated timespans by filtering and inspecting timespans from their input with respect to their voice name or music specifier label. This behavior helps model how timespans used to generate phrasing for a pianist's pedaling voice can be derived by selecting only those timespans in a timespan inventory for entire ensemble which pertain to the pianist's right and left-hand voices, ignoring all others.

Consider the following timespan inventory, comprised of both overlapping and non-overlapping timespans created for the contexts “Voice 1”, “Voice 2” and “Ignored Voice”:
A dependent timespan-maker can be configured to depend on timespans created for the “Voice 1” and “Voice 2” contexts by specifying a tuple of voice names during instantiation. Passing the previously defined timespan inventory as an argument when calling the dependent timespan-maker adds the newly-created dependent timespans to it. For the sake of brevity, and because timespan-makers modify any timespan inventory passed as an argument to `__call__()` in-place, we pass a copy of this timespan inventory, created via a call to `new()`, instead of the original:

```python
>>> music_specifiers = {'Dependent Voice': None}
>>> dependent_timespan_maker = consort.DependentTimespanMaker(
...     voice_names=(
...         'Voice 1',
...         'Voice 2',
...     ),
... )
>>> result = dependent_timespan_maker(
...     music_specifiers=music_specifiers,
...     timespan_inventory=new(timespan_inventory),
... )
>>> show(result, key='voice_name')
```
Note how the above timespan inventory shows the timespans for “Dependent Voice” outlining the start and stop offset for partitioned shards of the timespans for “Voice 1” and “Voice 2”, but ignoring the boundaries outlined by the timespans for the “Ignored Voice” context. Dependent timespans first select all timespans from their input timespan inventory matching their voice-names criteria, then partition them into shards in order to determine which offsets to use for timespan creation. Partitioning guarantees that the created dependent timespans do not exceed the bounds of the timespans they depend upon.

Configuring the dependent timespan-maker with its `include_inner_starts` flag set to true causes it to create contiguous groups of dependent timespans, as though splitting at every moment when a timespan it depends upon starts. Note that new dependent timespans begin at the offsets 10/1, 15/1 and 25/1, for example. This is because a new timespan begins in “Voice 1” at both 10/1 and 25/1, and another new timespan begins in “Voice 2” at 15/1:

```python
>>> new_dependent_timespan_maker = new(
    ...     dependent_timespan_maker,
    ...     include_inner_starts=True,
    ... )
```

```output
Dependent Voice:
   0  10  15  25  35  40  45  50  75

Ignored Voice:
   5  20  30  45  50  80

Voice 1:
   0  10  20  25  30  50  60  75

Voice 2:
   0  10  15  35  40  45  75
```

Likewise, the `include_inner_stops` flag causes the dependent timespan-maker to take into account the stop offsets of any timespan it depends upon:
Dependent timespan-makers are capable of performing simple transformations on the offsets they extract from their input timespan inventory. Rotation allows the timespan-maker to rotate the durations outlined by the offsets extracted from the timespans they select. Specifying a rotation index of 1 causes each created group of dependent
timespans to rotate its internal durations by that index. The 10-duration dependent timespan which previously occurred first in the “Dependent Voice” now occurs second, following a $\frac{5}{1}$-duration timespan:

```python
>>> rotated_dependent_timespan_maker = new(
...     new_dependent_timespan_maker,
...     rotation_indices=(1, ),
... )
>>> result = rotated_dependent_timespan_maker(
...     music_specifiers=music_specifiers,
...     timespan_inventory=new(timespan_inventory),
... )
>>> show(result, key='voice_name', range_=(0, 75))
```

![Diagram of dependent voice, ignored voice, voice 1, and voice 2 after rotation.]

More than one rotation index can be specified, allowing for each group of dependent timespans – as created from each shard of the partitioned selected timespans – to be rotated independently:

```python
>>> rotated_dependent_timespan_maker = new(
...     new_dependent_timespan_maker,
...     rotation_indices=(1, -1),
... )
>>> result = rotated_dependent_timespan_maker(
...     music_specifiers=music_specifiers,
...     timespan_inventory=new(timespan_inventory),
... )
>>> show(result, key='voice_name', range_=(0, 75))
```

![Diagram of dependent voice, ignored voice, voice 1, and voice 2 after rotation with multiple indices.]
Note that in the previous dependent timespan-maker examples no target timespan was specified. When passed a non-empty timespan inventory during calling, timespan-makers can treat that inventory’s timespan as their target timespan if no target timespan was specified explicitly. However, passing a target timespan to a dependent timespan-maker causes that timespan-maker to perform a logical AND of the target timespan with any selected timespans in the input inventory. Here the dependent timespans are constrained between the offsets $\frac{17}{1}$ and $\frac{58}{1}$:

```python
>>> result = new_dependent_timespan_maker(
...    music_specifiers=music_specifiers,
...    target_timespan=timespantools.Timespan(17, 58),
...    timespan_inventory=new(timespan_inventory),
...)
>>> show(result, key='voice_name', range_==(0, 75))

Dependent Voice:

Ignored Voice:

Voice 1:

Voice 2:

3.5 Rhythm-makers

Abjad’s rhythm-makers, like Consort’s timespan-makers, are highly-configurable factory classes which behave like partially evaluated functions, taking as input sequences of divisions – positive, non-reduced fraction tokens representing the divisions in some phrase of music – and producing selections of score components as output. Abjad’s rhythmmakertools library contains a variety of such classes, each providing a different strategy for rhythm generation, but unified by the same callable interface. Additionally, rhythmmakertools provides a collection of specifier classes which group related configuration values together for controlling the behavior of ties, beams, duration spelling and other notational aspects of each rhythm-maker’s output. Like many other classes in Abjad – timespans, for example – both these specifiers and the rhythm-makers themselves can be templated via calls to new(). A tour of these rhythm-maker classes demonstrates how a wide range of rhythmic textures can be produced.

---

9Rather than coercing input into sequences of Duration objects, which reduce their denominators as much as possible, rhythm-makers treat all input as non-reduced fractions, allowing them to disambiguate $\frac{4}{16}$ from $\frac{2}{8}$ or $\frac{6}{8}$ from $\frac{3}{4}$ and to therefore treat those division tokens as distinct.
3.5.1 Note rhythm-makers

Note rhythm-makers, arguably the simplest class of rhythm-maker, take a sequence of input divisions and “fill” them with notes, tied as necessary, such that the duration of each logical tie in each output division equals the duration of each input division:

```python
>>> note_rhythm_maker = rhythmmakertools.NoteRhythmMaker()
>>> divisions = [(3, 8), (4, 8), (3, 16), (4, 16), (5, 8), (2, 4)]
>>> show(note_rhythm_maker, divisions=divisions)
```

Like many objects implemented in Abjad and its extensions, rhythm-makers can be illustrated via a call to `show()`. Rhythm-maker illustrations take an optional `divisions` argument, specifying what durations should be used for the generated rhythmic output, grouping each of those divisions into measures for ease of visualization.

Rhythm-makers can be configured with a variety of specifiers, allowing for optional customization of their rhythm-generating behavior. For example, a `TieSpecifier` can be used to force a rhythm-maker to tie the last note of each output division to the first note of the next output division. When used with a note rhythm-maker, this effectively ties all notes in the output together:

```python
>>> note_rhythm_maker = new(
...     note_rhythm_maker,
...     tie_specifier=rhythmmakertools.TieSpecifier(
...         tie_across_divisions=True,
...     ),
... )
>>> show(note_rhythm_maker, divisions=divisions)
```

Other rhythm-maker specifiers influence beaming, tuplet spelling, or can cause a rhythm-maker to convert patterned groups of leaves in its output from notes to rests or vice versa. Rhythm-makers configured with `output masks` replace the contents of their output divisions with rests in a patterned per-division basis. A sequence of one or more `BooleanPattern` instances control the masking pattern. These patterns partition the rhythm-maker’s output divisions into segments of a given period, and then mask out divisions specified by indices within that period.
For example, a note rhythm-maker configured with a single output mask of a period of length 2 and a masking index 0 will cause the first of every two divisions to be silenced:

```python
>>> mask = rhythmmakertools.BooleanPattern(indices=[0], period=2)
>>> note_rhythm_maker = rhythmmakertools.NoteRhythmMaker(output_masks=[mask])
>>> show(note_rhythm_maker, divisions)
```

Reducing the period of the boolean pattern from 2 to 1 silences every output division, effectively turning a note-generating rhythm-maker into a rest-generating rhythm-maker:

```python
>>> mask = rhythmmakertools.BooleanPattern(indices=[0], period=1)
>>> note_rhythm_maker = rhythmmakertools.NoteRhythmMaker(output_masks=[mask])
>>> show(note_rhythm_maker, divisions)
```

### 3.5.2 Talea rhythm-makers

*Talea* rhythm-makers, like talea timespan-makers, create rhythmic output through the use of a talea – an infinitely cyclic pattern of durations. Talea rhythm-makers fill their output divisions with durations from their talea, splitting those durations across division boundaries when the divisions are over-full. The following trivial talea rhythm-maker uses a length-1 talea comprised of a single \(\frac{1}{16}\) duration:

```python
>>> talea_rhythm_maker = rhythmmakertools.TaleaRhythmMaker(
...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     ...     )
>>> show(talea_rhythm_maker, divisions)
```
Extending the talea counts to a sequence of alternating $\frac{1}{16}$ and $\frac{1}{8}$ notes produces more complex results. Note how the $\frac{1}{8}$ durations break over the boundaries of the $\frac{3}{16}$ measure, but remain tied together. Talea rhythm-makers handle talea splitting and tying transparently:

```ruby
>>> talea_rhythm_maker = new(
...    talea_rhythm_maker,
...    talea_counts=[1, 2],
...)
>>> show(talea_rhythm_maker, divisions=divisions)
```

With the talea counts changed to a descending sequence of durations, the pattern of split and tied durations also changes:

```ruby
>>> talea_rhythm_maker = new(
...    talea_rhythm_maker,
...    talea_counts=[4, 3, 2, 1],
...)
>>> show(talea_rhythm_maker, divisions=divisions)
```

Talea rhythm-makers can be configured to treat input divisions as though they had more counts than they actually do via their `extra_counts_per_division` property, causing tuplets to appear in the output. The following talea, reconfigured from the previous, adds an extra count to every second and third input division. This causes the $\frac{4}{8}$ and $\frac{3}{16}$ as well as the $\frac{5}{8}$ and $\frac{2}{4}$ divisions to become tuplets, each with a pre-prolated contents duration $\frac{1}{16}$ longer than their prolated duration. In other words, the $\frac{4}{8}$ division contains a tuplet whose contents sum to $\frac{9}{16}$, but scaled into a duration of $\frac{8}{16}$. Likewise, the $\frac{3}{16}$ division contains a tuplet whose contents sum to $\frac{4}{16}$, but scaled into a duration of $\frac{3}{16}$. Note how this tupletting causes the pattern of split and tied talea durations to shift. In the previous example, the second instance of the talea's $\frac{3}{16}$ duration occurred entirely during the $\frac{3}{16}$ division. Here, that same $\frac{3}{16}$ talea duration begins on the final $\frac{1}{16}$ of the $\frac{4}{8}$ division’s 9:8 tuplet, tied into the 4:3 tuplet in the following division:

121
Like note rhythm-makers, talea rhythm-makers can be configured with a tie specifier, causing the last note of each division to be tied to the first note of the next division. In note rhythm-makers, the contents of each non-masked division is guaranteed to be a single logical tie. Therefore, tying across divisions results in the entire output of the note rhythm-maker merging into the same logical tie. The output divisions of talea rhythm-makers generally contain more than one logical tie, and therefore tying across divisions tends to produce the effect of elided downbeats with intermittent attacks:

Talea rhythm-makers can also be configured to produce intermittent silences, either by specifying negative count values in the rhythm-maker’s talea, or by configuring the rhythm-maker with a special *burnish* specifier which casts logical ties generated by the rhythm-maker as either notes or rests, in a patterned way. Here, the 3-count in the talea rhythm-maker’s talea is changed to -3, resulting in the production of $\frac{3}{16}$-duration silences:
Alternatively, configuring the rhythm-maker with the following `BurnishSpecifier` allows the first logical tie of every other division to be converted to rests:

```python
>>> talea_rhythm_maker_with_rests = new(
...    talea_rhythm_maker,
...    burnish_specifier=rhythmmakertools.BurnishSpecifier(
...        left_classes=[Rest],
...        left_counts=[1, 0],
...    ),
...)
>>> show(talea_rhythm_maker_with_rests, divisions=divisions)
```

3.5.3 Incised rhythm-makers

*Incised* rhythm-makers behave similarly to note rhythm-makers, but allow for *incising* patterned sequences of notes and rests from the beginnings and ends of each output division, or even from the beginning and end of the entire sequence of divisions – the rhythm-maker *output*. Configuring an incised rhythm-maker to perform incision requires an `InciseSpecifier` instance. An unconfigured incised rhythm-maker behaves identically to an unconfigured note rhythm-maker:

```python
>>> incised_rhythm_maker = rhythmmakertools.IncisedRhythmMaker()
>>> show(incised_rhythm_maker, divisions=divisions)
```

Incise specifiers define talea and group counts for prefix and suffix incision. The talea – as determined by the numerators given in the `prefix_talea` and `suffix_talea` sequences combined with the `talea_denominator` – define the durations to be selected from, as well as the *sign* of the component to be incised. Negative talea items indicate rests, while positive talea items indicate notes. The count properties – `suffix_counts` and `prefix_counts` – indicate how many talea durations should be selected at each pass. The following incised rhythm-maker incises a single $\frac{1}{16}$ rest at the end of each division:

```python
>>> incised_rhythm_maker = rhythmmakertools.IncisedRhythmMaker(
...    incise_specifier=rhythmmakertools.InciseSpecifier(
...        prefix_talea=[3, 1],
...        prefix_counts=[1, 0],
...        suffix_talea=[3, 5],
...        suffix_counts=[1, 0],
...    ),
...)
>>> show(incised_rhythm_maker, divisions=divisions)
```
Extending the suffix talea results in a more complex incision pattern:

Prefix incision occurs in an identical fashion, by specifying sequences of group counts and talea numerators. Here, alternating groups of notes of length-1 and length-2 are incised from the beginning of each division:

Setting the incise specifier’s `fill_with_notes` property to `false` causes the rhythm-maker to fill the unincised portions of its output divisions with rests rather than notes. Filled mainly with silence, the incision pattern becomes much clearer:
Incised rhythm-makers can be configured to only incise the outer divisions of their output. The following rhythm-maker cuts $\frac{3}{8}$ rests from the beginning of the first division in its output, and the end of the last division:

```python
>>> incised_rhythm_maker = rhythmmakertools.IncisedRhythmMaker(
...     inciseSpecifier=rhythmmakertools.InciseSpecifier(
...         outer_divisions_only=True,
...         prefix_counts=[1],
...         prefix_talea=[-1],
...         suffix_counts=[1],
...         suffix_talea=[-1],
...         talea_denominator=8,
...     ),
... )
>>> show(incised_rhythm_maker, divisions=divisions)
```

Incised rhythm-makers also respect many of the same rhythm-maker specifiers as the other rhythm-maker classes defined in Abjad’s rhythmmakertools library. Like note rhythm-makers, they can be configured to tie the last note of each output division to the first note of the next output division via an instance of the TieSpecifier class:

```python
>>> incised_rhythm_maker = new(
...     incised_rhythm_maker,
...     tieSpecifier=rhythmmakertools.TieSpecifier(
...         tie_across_divisions=True,
...     ),
... )
>>> show(incised_rhythm_maker, divisions=divisions)
```

3.5.4 Even-division rhythm-makers

*Even-division* rhythm-makers attempt to divide each input division into runs of notes with a basic duration $1/denominator$, where the denominator is specified on a per-division basis by the rhythm-maker’s configurable denominator.
sequence property. When an input division does not exactly fit some multiple of this basic duration, some combination of augmentation or tupletting will be used to produce notes with durations as close to that basic duration as possible:

```python
>>> even_division_rhythm_maker = rhythmmakertools.EvenDivisionRhythmMaker()
>>> show(even_division_rhythm_maker, divisions=divisions)
```

Extending the even-division rhythm-maker’s denominator sequence to a 2-length pattern produces divisions filled alternatingly with $\frac{1}{8}$ and $\frac{1}{16}$ notes. Note that the $\frac{3}{16}$ measure is filled with a dotted $\frac{1}{8}$ note:

```python
>>> even_division_rhythm_maker = new(
...     even_division_rhythm_maker,
...     denominators=[8, 16],
... )
>>> show(even_division_rhythm_maker, divisions=divisions)
```

Extending the denominator sequence even further results in more complex output. Note here how the $\frac{5}{8}$ division is rendered as a tuplet:

```python
>>> even_division_rhythm_maker = new(
...     even_division_rhythm_maker,
...     denominators=[8, 4, 16],
... )
>>> show(even_division_rhythm_maker, divisions=divisions)
```

Like the talea rhythm-maker, even-division rhythm-makers can be configured to add extra counts to each output division, forcing the rhythm-maker to treat input divisions as longer than they actually are. Artificially lengthened divisions are rendered as tuplets. With the following rhythm-maker which attempts to fill each output division with a run of $\frac{1}{8}$ notes, every other division is extended by one count, causing the $\frac{3}{8}$ and $\frac{7}{4}$ divisions to be rendered as $5:4$ tuplets, and the $\frac{9}{16}$ division to be rendered as a $3:2$ tuplet:
When configured with a tie specifier, the even-division rhythm-maker obscures the downbeat of every output division:

And when reconfigured with the earlier 8-4-16 denominator pattern, the even-division rhythm-maker produces rich tupletted rhythmic output:

3.5.5 Composite rhythm-makers

Consort provides a class for aggregating multiple rhythm-maker instances together into a composite rhythm-maker, which applies its aggregated rhythm-makers conditionally against input divisions to generate rhythmic output. The following contrived composite rhythm-maker uses the previously note rhythm-maker – which only generates rests
– for the last of any sequence of input divisions, the previously defined incised rhythm-maker for the first of any sequence of input divisions, and the previously defined talea rhythm-maker for all other input divisions:

```python
>>> composite_rhythm_maker = consort.CompositeRhythmMaker(
    ...    default=talea_rhythm_maker,
    ...    last=note_rhythm_maker,
    ...    first=incised_rhythm_maker,
    ... )
>>> show(composite_rhythm_maker, divisions=divisions)
```

![Composite rhythm maker example](image)

3.6 Meter

Abjad models meter as a rhythm-tree of nested, durated nodes which outline a series of strongly and weakly accented offsets. The accent strength of a particular offset found in a meter’s rhythm-tree derives from the number of nodes in that tree sharing that offset as a start or stop. The more nodes in the rhythm-tree which share an offset, the greater the weight – the accentedness – of that offset is taken to be. Abjad can construct the rhythm tree for any meter from a numerator / denominator pair such as a rational duration or time signature. Meter construction involves the progressive division of the numerator of the input pair into groups of two and threes\(^{10}\), and the decomposition of any other prime factors into groups of threes and twos. Division by two always occurs before division by three, giving preference to even metrical structures above odd or otherwise prime divisions. Constructing rhythm-trees in this fashion gives results which generally align with common practice expectations.

Consider the following \(\frac{6}{8}\) meter and its graph representation:

```python
>>> six_eight_meter = metertools.Meter((6, 8))
>>> graph(six_eight_meter)
```

![Six eighth meter graph](image)

\(^{10}\)The factors 4 and 5 are also used in meter rhythm-tree generation as they provide better typical results during meter rewriting.
The triangular and rectangular boxes indicate nodes in the rhythm-tree itself. Rectangular boxes represent “beats” – the leaves of the rhythm-tree – while triangular boxes indicate larger metrical groupings. The ovals at the bottom of the graph indicate – at their top – the start or stop offset of the nodes connected to them from above and – at their bottom – the relative weight of their accent. The final oval on the right indicates the offset and accent weight of the “next” downbeat.

The topmost triangle in the above graph represents the “highest” metrical grouping in a \( \frac{6}{8} \) meter. Tracing the leftmost and rightmost arrows down through the topmost node’s children gives the offsets 0 and \( \frac{3}{4} \): the first downbeat and next downbeat in a \( \frac{6}{8} \) meter. Offsets 0 and \( \frac{3}{4} \) also have the strongest accent weights as they occur as either the start offset or stop offset of nodes at three levels of hierarchy in the rhythm tree. At the second level the \( \frac{6}{8} \) grouping divides into two \( \frac{3}{8} \) groupings, following common practice expectations: metrical groupings tend to subdivide into groups of two before they subdivide into groups of three.\(^1\) Both second-level nodes share the offset of \( \frac{3}{8} \), which also occurs in the third level, giving \( \frac{3}{8} \) a weight of two. The third level contains the \( \frac{1}{8} \) duration beats, grouped by their parents in the second level into two groups three \( \frac{1}{8} \) duration nodes. The offsets \( \frac{1}{8}, \frac{1}{4}, \frac{1}{2} \) and \( \frac{5}{8} \) are not shared by any nodes except at the lowest metrical level and therefore all receive an accent weight of one.

Consider the following other examples of meters modeled in Abjad. A \( \frac{3}{4} \) meter consists of a top-level \( \frac{3}{4} \) metrical grouping divided into three \( \frac{1}{4} \) duration beats:

```
>>> three_four_meter = metertools.Meter((3, 4))
>>> graph(three_four_meter)
```

![Diagram of a \( \frac{3}{4} \) meter]

By default, a \( \frac{7}{8} \) meter subdivides its top-level metrical grouping into \( \frac{3}{8}+\frac{2}{8}+\frac{2}{8} \) groupings:

```
>>> seven_eight_meter = metertools.Meter((7, 8))
>>> graph(seven_eight_meter)
```

\(^1\)Consider a \( \frac{12}{8} \) meter. Western musicians tend to subdivide twelve into either two groups of six or four groups of three rather than into three groups of four.
A $12/8$ meter subdivides into four $3/8$ duration groupings, each containing three $1/8$ duration beats:

```
>>> twelve_eight_meter = metertools.Meter((12, 8))
>>> graph(twelve_eight_meter)
```

Abjad also permits alternate representations of meters which ostensibly share the same numerator and denominator. The default interpretation of $4/4$ generates a top-level rhythmic grouping with a duration of $4/4$ and four $1/4$ beats as children:

```
>>> four_four_meter = metertools.Meter((4, 4))
>>> graph(four_four_meter)
```

While meter objects are usually instantiated from numerator / denominator pairs, with their rhythm-tree structure determined programmatically from that input pair, they can also be instantiated from strings parsable as rhythm-trees, or from rhythm-tree objects themselves. All meters, because they are implemented in terms of rhythm-trees, can be represented by a Lisp-like rhythm-tree syntax:

---

12 A “flat” $4/4$ metrical structure is useful for meter rewriting as it allows the meter rewriting algorithm to ignore many common rhythmic idioms like $1/4+1/2+1/4$ and $1/4+1/4$. 
Instantiating meters from explicit rhythm tree syntax allows composers to choose alternate representations of metrical structures. For example, a $\frac{4}{4}$ meter which strongly emphasizes beat three is possible by subdividing the top-level $\frac{4}{4}$ metrical grouping into two $\frac{2}{4}$ duration groupings, which are then subdivided each into two $\frac{1}{4}$ duration beats. Such a metrical structure effectively treats $\frac{4}{4}$ as identical to $\frac{3}{2}$:

Unusual metrical structures are also possible, such as the following $\frac{4}{4}$ meter which divides into two parts, with the first part dividing into two again, and the second grouping of that divided into two again:

3.7 Rewriting meters

Notated rhythms can be expressed in multiple ways while maintaining the same attack-point and duration structure. *Meter rewriting* formalizes the process of re-notating a rhythm according to the offset structure inherent to some
meter while maintaining the original attack-points and durations by fusing and splitting logical ties according to their *validity*. In the context of meter rewriting, validity expresses whether any logical tie – trivial or not – *aligns* to offsets found in nodes at a particular depth in a meter’s rhythm-tree. Alignment requires that a given logical tie either starts or stops at offsets found in the collection of offsets defined by a subtree of nodes in a meter’s rhythm-tree. Meter rewriting proceeds by testing logical ties against offsets outlined first by the root node of a meter and, if found invalid, against those offsets found in nodes progressively deeper in the meter’s rhythm-tree.

For example, the offsets outlined by the root node of a $6/8$ meter can be found by examining its depth-wise offset inventory:

```python
>>> six_eight_meter = metertools.Meter((6, 8))
>>> six_eight_meter.depthwise_offset_inventory[0]
(Offset(0, 1), Offset(3, 4))
```

A logical tie three-quarters in duration starting at the offset 0 would be considered valid in the context of $6/8$ because it aligns to the offsets outlined at depth-0 of the meter’s rhythm-tree. In contrast, a logical tie two quarters in duration starting at $1/8$ – therefore outlining the timespan of $1/8$–$3/8$ – would be considered invalid due to misalignment:

```python
>>> six_eight_measure = Measure((6, 8), "r8 c'2 r8")
>>> show(six_eight_measure)
```

Meter rewriting splits misaligned logical ties at any possible offset found in the currently considered depth. If no offsets at that depth intersect with the misaligned tie, the depth is increased and the process repeats. At a depth of 1 in a $6/8$ meter we finally find an offset intersecting the timespan of the misaligned $1/8$–$3/8$ logical tie at the offset $3/8$:

```python
>>> six_eight_meter.depthwise_offset_inventory[1]
(Offset(0, 1), Offset(3, 8), Offset(3, 4))
```

Rewriting the contents of the $6/8$ measure against a $6/8$ meter splits the inner half-note at the $3/8$ offset boundary:

```python
>>> mutate(six_eight_measure).rewrite_meter(six_eight_meter)
>>> show(six_eight_measure)
```

As another example, consider the following rhythm:
The middle measure is notated in a perfectly valid manner. However, the double-dotted D does not align with or break against any of the offsets of a \(4/4\) metrical structure: \(0/4\), \(1/4\), \(2/4\), \(3/4\) or \(4/4\):

```python
>>> four_four_meter = metertools.Meter((4, 4))
>>> graph(four_four_meter)
```

Rewriting the inner measure against a \(4/4\) meter breaks the inner logical tie at \(1/4\) and \(3/4\), slightly improving readability:

```python
>>> mutate(staff[1]).rewrite_meter(four_four_meter)
>>> show(staff)
```

Rewriting the same measure against a \(2/2\) meter breaks the inner logical tie at the offset \(1/2\), as two double-dotted quarter notes. While identical to \(4/4\) in duration, \(2/2\) strongly emphasizes this duple division:

```python
>>> two_two_meter = metertools.Meter((2, 2))
>>> staff = Staff(parseable)
>>> mutate(staff[1]).rewrite_meter(two_two_meter)
>>> show(staff)
```

### 3.7.1 Dot count

Meter rewriting can control for various qualities of how rhythms are notated. For example, the maximum number of dots allowed for any notated rhythmic value can be constrained. Logical ties encountered during the rewriting
process whose individual notes exceed the maximum number of permitted dots will be rewritten, if a maximum dot count has been specified.

Consider this series of progressively rewritten rhythms, beginning with the following unrewritten \( \frac{3}{4} \) measure:

```python
>>> measure = Measure((3, 4), "c'32 d'8 e'8 fs'4..."
>>> show(measure)
```

After meter rewriting, the final F-sharp is still notated as a triple-dotted quarter-note, valid because its stop offset aligns perfectly with the containing measure’s stop offset:

```python
>>> mutate(measure).rewrite_meter((3, 4))
>>> show(measure)
```

Capping the maximum number of dots to 2 causes the F-sharp to be rewritten as a double-dotted eighth-note tied to a quarter-note:

```python
>>> measure = Measure((3, 4), "c'32 d'8 e'8 fs'4..."
>>> mutate(measure).rewrite_meter((3, 4), maximum_dot_count=2)
>>> show(measure)
```

Constraining the maximum number of dots to 1 further subdivides the F-sharp logical tie:

```python
>>> measure = Measure((3, 4), "c'32 d'8 e'8 fs'4..."
>>> mutate(measure).rewrite_meter((3, 4), maximum_dot_count=1)
>>> show(measure)
```

Finally, with no dots permitted at all, the rhythmic presentation of the measure changes considerably. Every dotted rhythm has been subdivided:

```python
>>> measure = Measure((3, 4), "c'32 d'8 e'8 fs'4..."
>>> mutate(measure).rewrite_meter((3, 4), maximum_dot_count=0)
>>> show(measure)
```

Prudent application of dot count constraints can clarify awkward rhythmic spellings by forcing the appearance of offsets in logical ties inherent to the rewriting meter.
3.7.2 Boundary depth

In meter rewriting, *boundary depth* forces emphasis of offsets found at different *depths* in a meter's rhythm-tree by marking as invalid logical ties which do not immediately align to those offsets on the very first pass of the recursive meter rewriting process. Use of boundary depth often clarifies distinctive groupings in more deeply nested meters, such as the “3-ness” inherent to a $\frac{9}{8}$ meter. The nodes with a depth of 1 in the following $\frac{9}{8}$ meter’s rhythm-tree – the three $\frac{3}{8}$ inner nodes marked by triangles – outline the offsets $\frac{9}{8}$, $\frac{3}{8}$, $\frac{6}{8}$ and $\frac{9}{8}$. When rewriting a $\frac{9}{8}$ rhythm with a boundary depth of 1, any logical ties not aligning with – either starting or stopping at – those offsets would be marked as invalid and therefore rewritten:

```python
>>> nine_eight_meter = metertools.Meter((9, 8))
>>> graph(nine_eight_meter)
```

```plaintext
\[\text{9/8} \quad 3/8 \quad 3/8 \quad 3/8 \quad 1/8 \quad 1/8 \quad 1/8 \]
```

```python
>>> measure = Measure((9, 8), "c'2 d'2 e'8")
>>> show(measure)
```

```plaintext
\[\text{C \ D \ E} \]
```

```python
>>> mutate(measure).rewrite_meter(nine_eight_meter)
>>> show(measure)
```

```plaintext
\[\text{C \ D \ E} \]
```

After rewriting, without any boundary depth specified, the D half-note in the above $\frac{9}{8}$ measure has been split into two quarter-notes, tied together. The second of these quarter-notes begins at $\frac{6}{8}$, therefore aligning with the start of the third $\frac{3}{8}$ node at depth 1 in the $\frac{9}{8}$ meter’s rhythm tree. Likewise, the first half of the split half-note ends at the same $\frac{6}{8}$ offset. However, the initial C half-note, while aligning with the beginning of the meter and therefore treated as valid, does not emphasize any of the meter’s inner offsets as outlined by the offsets $\frac{3}{8}$ and $\frac{6}{8}$:
After rewriting with boundary depth set to 1, not only has the D half-note been split in half, but the initial C half-note has been split at its $3\circ8$ offset. Because the initial half note aligned at the $9\circ8$ offset – as outlined by the root node in the $9\circ8$ meter’s rhythm tree –, but not at the $3\circ8$ offset – as outlined by the first and second $3\circ8$-duration nodes at depth 1 of the same meter’s rhythm tree –, it was marked invalid and therefore split at the first available offset: $3\circ8$.

3.7.3 **Recursive meter rewriting**

Meter rewriting treats the contents of tuplets with non-trivial prolation as existing within their own metrical scope, isolated from any other meter. The numerator and denominator of the tuplet’s pre-prolated contents duration act as the numerator and denominator of their “virtual” meter. Thus, a 6:5 tuplet encountered in any context will be rewritten as though under some 6-numerator meter:

If, while rewriting the contents of one tuplet, a second tuplet is encountered as a child of that first tuplet, the meter rewriting algorithm will recursively descend into that second tuplet – and any further tuplet children at any depth. This recursive descent allows any encountered component to be rewritten in a relevant metrical context.
3.7.4 Examples

Meter rewriting can clarify structural differences between meters with identical durations, such as $\frac{3}{4}$ and $\frac{6}{8}$, or between various possible representations of other prime-numerator meters like $\frac{3}{4}$, $\frac{7}{8}$ and so forth. The following series of $\frac{3}{4}$ measures contain rhythms which emphasize $\frac{3}{4}$-ness or $\frac{6}{8}$-ness to different degrees. While perfectly valid and totally legible, they can still be rewritten to more strongly express one meter over the other:

```
>>> staff = Staff(context_name='RhythmicStaff')
>>> staff.extend("{ c'2 c'4 } { c'4. c'4. } { c'2 ~ c'8 c'8 }")
>>> attach(TimeSignature((3, 4)), staff)
>>> show(staff)
```

Rewriting under a $\frac{3}{4}$ meter with a boundary depth of 1 forces emphasis of the offsets found in the $\frac{3}{4}$ meter's depth-1 group of three quarter-duration nodes: $\frac{0}{4}$, $\frac{1}{4}$, $\frac{2}{4}$ and $\frac{3}{4}$. Both the outer measures’ contents align to these offsets, but the inner measure’s hemiola contents do not, and are rewritten:

```
>>> for container in staff:
...     mutate(container).rewrite_meter((3, 4), boundary_depth=1)
... >>> show(staff)
```

Conversely, rewriting with a $\frac{6}{8}$ meter – again with a boundary depth of 1 – forces emphasis of the offsets outlined by $\frac{6}{8}$’s two depth-1 $\frac{3}{8}$-duration nodes: $\frac{0}{8}$, $\frac{3}{8}$ and $\frac{6}{8}$. Only the inner measure’s contents align perfectly these offsets, so the outer measures are rewritten, better demonstrating $\frac{6}{8}$’s duple-ness:

```
>>> staff = Staff(context_name='RhythmicStaff')
>>> staff.extend("{ c'2 c'4 } { c'4. c'4. } { c'2 ~ c'8 c'8 }")
>>> attach(TimeSignature((6, 8)), staff)
>>> for container in staff:
...     mutate(container).rewrite_meter((6, 8), boundary_depth=1)
... >>> show(staff)
```

3.8 Finding meters

Not only can meters be used to alter rhythmic structures, they can also be derived from them. A meter’s weighted-offset pattern can be used as one-dimensional kernel, or convolution matrix, to determine how strongly an arbitrary
collection of offsets appears to express that meter. Given a collection of meters to choose from, each meter can be matched against those offsets and the meter most closely aligning selected as the winner. This process is called meter fitting.

3.8.1 Offset counters

Before convolving a meter with a collection of offsets, those offsets need to be extracted and counted. Abjad’s metertools provides an OffsetCounter class which maps offsets against counts. Offset counters can be instantiated from any expression containing offsets or whose elements can be expressed as timespans and therefore possess both start and stop offsets. Offsets which appear multiple times in the input expression will result in a higher count in the offset counter, and will in turn have a greater influence during meter fitting.

Consider the following score example:

```python
>>> upper_staff = Staff("c'8 d'4. e'8 f'4."
>>> lower_staff = Staff(r'\clef bass c4 b,4 a,2')
>>> piano_staff = scoretools.StaffGroup(
...    [upper_staff, lower_staff],
...    context_name='PianoStaff',
...  )
>>> show(piano_staff)
```

The start and stop offsets of all of the leaves of this score can be counted by selecting the score’s leaves and instantiating an offset counter from them. Because all score components can be expressed as timespans via inspect_(some_component).get_timespan() the offset counter can retrieve both their start and stop offsets within the score:

```python
>>> leaves = piano_staff.select_leaves(allow_discontiguous_leaves=True)
>>> piano_staff_counter = metertools.OffsetCounter(leaves)
>>> print(format(piano_staff_counter))
metertools.OffsetCounter(
    {
        durationtools.Offset(0, 1): 2,
        durationtools.Offset(1, 8): 2,
        durationtools.Offset(1, 4): 2,
        durationtools.Offset(1, 2): 4,
        durationtools.Offset(5, 8): 2,
        durationtools.Offset(1, 1): 2,
    }
)
```
Note that the offset \( \frac{1}{2} \) shows a count of 4. This is because \( \frac{1}{2} \) acts as both the start or stop offset for four separate leaves in the score.

Offset counters can also be generated from timespan inventories, allowing meter convolution to be used without reference to any score objects at all:

```python
>>> timespans = timespantools.TimespanInventory([
    ...    timespantools.Timespan(-1, 10),
    ...    timespantools.Timespan(5, 15),
    ...    timespantools.Timespan(15, 20),
    ...    timespantools.Timespan(10, 15),
    ...])
>>> timespan_counter = metertools.OffsetCounter(timespans)
>>> show(timespan_counter)
```

3.8.2 Metric accent kernels

As demonstrated earlier, Abjad’s model of meter describes a sequence of offsets with varying degrees of weight – accentedness – attributed to each offset, as determined by the hierarchical tree structure of that meter. This model allows us to explain how downbeats have a stronger weight than upbeats, and how the \( \frac{3}{8} \) offset in a \( \frac{6}{8} \) measure is less strong than its downbeat but still stronger than the offsets at \( \frac{1}{8}, \frac{2}{8}, \frac{4}{8}, \) or \( \frac{5}{8} \). However, the default model of a \( \frac{6}{8} \) meter makes no explicit reference to offsets such as \( \frac{1}{16}, \frac{3}{16}, \) or \( \frac{7}{16} \). A common practice understanding of meter tells us that these offsets should be taken as less accented than those at \( \frac{1}{8}, \frac{2}{8}, \frac{4}{8}, \) and so forth. They effectively represent an even lower level of leaves on the rhythm-tree for \( \frac{6}{8} \). Likewise, offsets with a denominator of 32 or 64 should be explainable in an identical fashion.

Abjad’s `MetricAccentKernel` class provides an object model for both the act of progressively subdividing the weighted offsets of a meter down to some arbitrary denominator, and for the process of convolving those offsets as a one-dimensional convolution kernel against an offset counter. Metric accent kernels allow meters to be fitted against offset counters containing offsets not explicitly modeled by the rhythm-trees of those same meters.
Here a metric accent kernel is generated from a $\frac{4}{4}$ meter, extending its denominator to a limit of $\frac{1}{16}$. The weights at each offset are normalized such that they sum to 1. Normalization prevents very long meters from having undue influence during meter fitting:

```python
>>> meter = metertools.Meter((4, 4))
>>> kernel_44 = metertools.MetricAccentKernel.from_meter(meter, denominator=16)
>>> for offset, weight in sorted(kernel_44.kernel.items()):
...     print('{!s}	{!s}'.format(offset, weight))
...
0**I4/33
1/16**I1/33
1/8**I2/33
3/16**I1/33
1/4**I1/11
5/16**I1/33
3/8**I2/33
7/16**I1/33
1/2**I1/11
9/16**I1/33
5/8**I2/33
11/16**I1/33
3/4**I1/11
13/16**I1/33
7/8**I2/33
15/16**I1/33
1**I4/33
```

The $\frac{4}{4}$ metric accent kernel can be called against an offset counter – as though it were a function – to generate the convolution response. The count at each offset in the input offset counter is multiplied against the weight at the corresponding offset in the metric accent kernel. If no corresponding offset exists in the kernel, the weight is taken as 0. The weighted counts are then added together and returned:

```python
>>> response = kernel_44(piano_staff_counter)
>>> float(response)
0.5454545454545454
```

The following loop demonstrates the logic underlying the above meter convolution process example:

```python
>>> total = Multiplier(0, 1)
>>> for offset, count in sorted(piano_staff_counter.items()):
...     weight = Multiplier(0, 1)
...     if offset in kernel_44.kernel:
...         weight = kernel_44.kernel[offset]
...     weighted_count = weight * count
...     total += weighted_count
...     message = '{!s}:\tcount: {}, weight: {!s}, multiplied: {!s}, total: {!s}'
```
Now consider the metric accent kernels for $3/4$, $7/8$ and $5/4$ meters:

```python
>>> kernel_34 = metertools.MetricAccentKernel.from_meter((3, 4), denominator=16)
>>> kernel_78 = metertools.MetricAccentKernel.from_meter((7, 8), denominator=16)
>>> kernel_54 = metertools.MetricAccentKernel.from_meter((5, 4), denominator=16)
```

A convolution response can be generated for each of these kernels against the piano staff offset counter:

```python
>>> float(kernel_34(piano_staff_counter))
0.5384615384615384
>>> float(kernel_78(piano_staff_counter))
0.4482758620689655
>>> float(kernel_54(piano_staff_counter))
0.4186046511627907
```

Note that the previously recorded response for a $4/4$ meter is still higher than any of these three responses.

### 3.8.3 Meter fitting

Meter fitting involves the progressive comparison of a collection of permitted meters against the offsets and counts in an offset counter. Starting from the lowest offset, the metric accent kernel for each meter is convolved with all those offsets in the offset counter with which it overlaps. The response from each convolution is recorded and the meter with the highest associated response is selected to represent those offsets. The process repeats, starting at the right-most offset of the last meter selected, until no more offsets remain to compare against.

Consider the following offsets, which are separated from one another by the duration of a whole note, outlining the start and stop offsets of a series of $4/4$ meters:

```python
>>> offset_counter = metertools.OffsetCounter([
...     (0, 4), (4, 4), (8, 4), (12, 4), (16, 4),
...   ])
>>> show(offset_counter, range_=(0, 5))
```
Given the following collection of meters, after meter fitting is performed only $\frac{3}{4}$ meters should be selected as they perfectly align with the previously defined collection of offsets:

```python
>>> permitted_meters = metertools.MeterInventory([(3, 4), (4, 4), (5, 4)])
>>> show(permitted_meters, range=(0, 5))

[1 1 1 1 1]

>>> fitted_meters = metertools.Meter.fit_meters_to_expr(
... expr=offset_counter,
... meters=permitted_meters,
... )
>>> show(fitted_meters, range=(0, 5))

[1 1 1 1 1]

If we change the input offsets to no longer outline only $\frac{3}{4}$ meters, the meter fitting process will arrive at a different solution. Note how the following fitted meters emphasize the $\frac{3}{4}$ and $\frac{5}{4}$ durations inherent to the new input offsets. The initial $\frac{3}{4}$ fitted meter perfectly matches the offsets $\frac{3}{4}$ and $\frac{3}{4}$, while the $\frac{4}{4}$ meter after it matches the $\frac{5}{4}$ offset against its third beat. The next meter’s end-beat offset aligns against the offset counter’s $\frac{3}{2}$ offset allowing the meter fitting process to perfectly match the remaining offsets against a pair of $\frac{5}{4}$ meters:

```python
>>> offset_counter = metertools.OffsetCounter([... (0, 4), (3, 4), (5, 4), (10, 4), (15, 4), (20, 4), ...])
>>> fitted_meters = metertools.Meter.fit_meters_to_expr(
... expr=offset_counter,
... meters=permitted_meters,
... )
>>> show(offset_counter, range=(0, 5))

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

>>> show(fitted_meters, range=(0, 5))

[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```
Meter fitting can also control for how many times meters are permitted to immediately repeat. The above example ends with two $\frac{3}{4}$ meters in a row. If desired, this repetition can be prevented entirely by the `maximum_run_length` keyword to `meter.fit_meters_to_expr()`. With `maximum_run_length` set to 1, the meter fitting process can no longer select two $\frac{3}{4}$ meters for the end of the fitted meter sequence. Instead, a pair of $\frac{3}{4}$ and $\frac{4}{4}$ meters replace the previously final $\frac{3}{4}$ meter:

```python
def fitted_meters = metertools.Meter.fit_meters_to_expr(
    ...  expr=offset_counter,
    ...  meters=permitted_meters,
    ...  maximum_run_length=1,
    ... )

>>> show(fitted_meters, range_=(0, 5))
```

3.8.4 Examples

The output of timespan-makers motivates meter fitting more forcefully than the above trivial examples. Densely layered timespan textures often involve hundreds or even thousands of offsets with many points of simultaneity or overlap. Careful management of meter fitting can result in convincing metrical solutions to such textures.

Recall the quartet timespan texture from earlier in this chapter:

```python
>>> music_specifiers = collections.OrderedDict([  
    ...  ('Voice 1', None),
    ...  ('Voice 2', None),
    ...  ('Voice 3', None),
    ...  ('Voice 4', None),
    ... ])

>>> target_timespan = timespantools.Timespan(0, (19, 4))

>>> timespan_inventory = talea_timespan_maker(  
    ...  music_specifiers=music_specifiers,
    ...  target_timespan=target_timespan,
    ... )

>>> show(timespan_inventory, key='voice_name')
```
Constructing an offset counter from this timespan inventory shows several offsets with strong simultaneities, notably at 11/8, 13/8, 17/8 and 25/8:

```python
>>> offset_counter = metertools.OffsetCounter(timespan_inventory)
>>> show(offset_counter, range_=0, (19, 4))
```

As before, a collection of permitted meters are fit against the offset counter. Compare the weights in the counter to the meter sequence below it. Offsets in the counter with weights greater than 1 tend to have been fitted against strong beats in each meter. The initial 6/8 meter receive a 2-count accent on its half-way 3/8 offset. The second 6/8 meter matches strong accents on both its downbeat, its half-beat and on its end-beat. Likewise, the 7/8 meter matches strong accents on its down-beat and end-beat, as well as its inner 2-count groupings:

```python
>>> permitted_meters = metertools.MeterInventory([
...     (5, 8), (3, 4), (6, 8), (7, 8), (4, 4),
...])
>>> fitted_meters = metertools.Meter.fit_meters_to_expr(
...     expr=offset_counter,
...     meters=permitted_meters,
...     maximum_run_length=1,
...)
>>> show(offset_counter, range_=0, (19, 4))
```

```python
>>> show(fitted_meters, range_=0, (19, 4))
```
Removing the $\frac{5}{8}$ meter from the permitted meters inventory gives less convincing results. Notably, the strong simultaneities at offsets $\frac{11}{8}$ and $\frac{17}{8}$ no longer align with any downbeats in the fitted meters:

```python
>>> permitted_meters = metertools.MeterInventory([... (3, 4), (6, 8), (7, 8), (4, 4), ...])
>>> fitted_meters = metertools.Meter.fit_meters_to_expr(... expr=offset_counter,... maximum_run_length=1,... meters=permitted_meters,...)
>>> show(offset_counter, range_=0, (19, 4))
```

Lifting the constraint on meter repetition does not improve the results. Both the $\frac{11}{8}$ and $\frac{17}{8}$ offsets still align with weak beats in the fitted meters. The only real change is the swapping of $\frac{6}{8}$ and $\frac{3}{4}$ meters in the first and last pairs of fitted meters:

```python
>>> fitted_meters = metertools.Meter.fit_meters_to_expr(... expr=offset_counter,... meters=permitted_meters,...)
>>> show(offset_counter, range_=0, (19, 4))
```

However, reinstating the repetition constraint and permitting a greater variety of meters gives an even closer fitting than the very first example. Not only do the $\frac{11}{8}$ and $\frac{17}{8}$ offsets align at downbeats, but the three-weight offset at $\frac{25}{8}$ matches the downbeat of a $\frac{3}{4}$ meter with two-weight accents on both its second and third beat as well as the following downbeat:
permitted_meters = metertools.MeterInventory([
    ... (2, 4), (4, 8), (3, 4), (6, 8), (7, 8), (4, 4),
    ... ])

fitted_meters = metertools.Meter.fit_meters_to_expr(
    ... expr=offset_counter,
    ... maximum_run_length=1,
    ... meters=permitted_meters,
    ... )

show(offset_counter, range_=\((0, (19, 4))\))

>>> show(fitted_meters, range_=\((0, (19, 4))\))

Increasing the number of permitted offsets even further does not necessarily improve results further. A better initial
meter fitting – like the initial \(\frac{9}{8}\) here – often diminishes the choices possible afterward:

>>> permitted_meters = metertools.MeterInventory([
    ... (2, 4), (3, 16), (3, 4), (4, 4), (4, 8), (5, 16),
    ... (5, 4), (5, 8), (6, 8), (7, 8), (9, 8),
    ... ])

>>> poorly_fitted_meters = metertools.Meter.fit_meters_to_expr(
    ... expr=offset_counter,
    ... maximum_run_length=1,
    ... meters=permitted_meters,
    ... )

>>> show(offset_counter, range_=\((0, (19, 4))\))

>>> show(poorly_fitted_meters, range_=\((0, (19, 4))\))

The process of meter fitting described here is not perfect. While useful for creating generally convincing metrical
solutions to dense timespan structures, meter fitting could probably be better solved through human intervention.
The fitting process often over-emphasizes local attack point maxima while ignoring larger or more elegant metrical patterns. Likewise, the process often provides an accurate fitting for the very first meter, but causes a cascade of poor solutions for every following meter. Where procedural meter fitting wins over human intervention is in speed. Solutions to enormous timespan inventories with hundreds or thousands of timespans can be found in less than a second, facilitating the rapid sketching and revising of timespan-based musical structures.

Meter fitting can be considered as a kind of constraint problem. The set of permitted meters act as a search space while each progressive selection of offsets from the input offset counter act as the problem to solve against. In that light, improvements might involve searching for meter solutions with increased *lookahead* – that is, fitting the current meter based not only on its response to a selection of offsets, but also on how well any meter following it would score. Searching the offset counter for patterns, changing the weighting algorithm of metric accent kernels, or jumping directly to the attack point maxima in the input offset counter and solving forwards and backwards from them are also possible avenues for improvement.

### 3.9 Synthesizing time techniques

The techniques outlined in this chapter – timespan inventories, timespan-makers, rhythm-makers, meter fitting and rewriting – describe various ways of modeling, creating and manipulating aspects of musical score. Taken separately, none of them can ever result in the rhythmic framework for a polyphonic piece of music. However, by combining all of them together, it is possible to construct powerful tools for generating arbitrarily large amounts of notation. The following `build_score()` function sketches one possible approach to combining these techniques:

```python
def build_score(
    performed_rhythm_maker,
    permitted_meters,
    score_template,
    timespan_inventory,
):
    fitted_meters, meter_boundaries = get_meters_and_meter_boundaries(
        timespan_inventory, permitted_meters
    )
    all_voicewise_timespans = get_all_voicewise_timespans(timespan_inventory)
    seed = 0
    score = score_template()
    for voice in iterate(score).by_class(Voice):
        if voice.name not in all_voicewise_timespans:
            all_voicewise_timespans[voice.name] = \
                timespantools.TimespanInventory()
            voice_timespans = all_voicewise_timespans[voice.name]
            previous_stop_offset = Offset(0)
            for shard in voice_timespans.partition(include_tangent_timespans=True):
```
if shard.start_offset != previous_stop_offset:
    silent_music = make_silent_music(
        meter_boundaries=meter_boundaries,
        start_offset=previous_stop_offset,
        stop_offset=shard.start_offset,
    )
    voice.append(silent_music)
performed_music = make_performed_music(
    meter_boundaries=meter_boundaries,
    start_offset=previous_stop_offset,
    stop_offset=shard.start_offset,
)
    voice.append(performed_music)
    performed_music = make_performed_music(
        meter_boundaries=meter_boundaries,
        rhythm_maker=performed_rhythm_maker,
        seed=seed,
        timespans=shard,
    )
    voice.append(performed_music)
    performed_music = make_performed_music(
        meter_boundaries=meter_boundaries,
        rhythm_maker=performed_rhythm_maker,
        seed=seed,
        timespans=shard,
    )
    voice.append(performed_music)
    voice.append(silent_music)

seed += 1
previous_stop_offset = shard.stop_offset
if previous_stop_offset != meter_boundaries[-1]:
    silent_music = make_silent_music(
        meter_boundaries=meter_boundaries,
        start_offset=previous_stop_offset,
        stop_offset=meter_boundaries[-1],
    )
    voice.append(silent_music)
for phrase in voice:
    rewrite_meters(phrase, fitted_meters, meter_boundaries)
add_time_signature_context(score, fitted_meters)
return score

build_score() creates a score from a rhythm-maker, a sequence of permitted meters, a score template and a timespan inventory. The timespan inventory must contain performed timespans whose voice names align with the voice names in the score produced by the score template. For example, Abjad’s GroupedRhythmicStavesScoreTemplate class can produce scores containing voices with names like “Voice 1”, “Voice 2” and so forth. The timespan inventory used for this example – created earlier in this chapter – contains performed timespans with the voice names “Voice 1”, “Voice 2”, “Voice 3” and “Voice 4”, requiring a score template capable of creating scores with four staves of one voice each:

>>> show(timespan_inventory, key='voice_name')
Voice 1:

Voice 2:

Voice 3:

Voice 4:
Score building proceeds by first fitting the permitted meters against the timespan inventory and calculating their *meter boundaries* – the offsets of each downbeat, as well as the virtual downbeat, the “end-beat”, after the final meter. Then the timespans in the input timespan inventory are separated into new timespan inventories according to their associated voice names, with these new inventories stored in a dictionary whose keys are those same voice names. With the timespan and metrical structures organized, the `build_score()` produces an unpopulated score from its score template. For each voice in this score, an inventory of timespans is retrieved from the `all_voicewise_timespans` dictionary. If no inventory exists, an empty one is created. The associated performed timespans are partitioned into contiguous groups in order to facilitate rhythm generation. Each partitioned group is passed, along with the input rhythm-maker, the fitted meter boundaries, and a seed value – the number of performed timespan groups encountered so far – to a rhythm generation subroutine, with the resulting phrase container appended into the current voice. If a gap is encountered between two partitioned groups, between the beginning of the first partitioned group and the beginning of the score – the offset 0, or between the end of the last partitioned group and the final meter boundary, “silent music” will be created in the form of rests grouped into a container, with that
container then inserted into the current voice. Once all voices in the score have been populated, their meters are rewritten, then the score is formatted and finally returned.

An analysis of each of the functions called within `build_score()` follows.

### 3.9.1 Organizing meter

Consider the first function called inside `build_score()`, `get_meters_and_meter_boundaries()`. A meter sequence of equal or greater duration to the input timespan inventory can be produced through fitting a collection of permitted meters against that same input timespan inventory. The sequence of offsets found at the boundaries of each fitted meter can then be determined by computing the cumulative sums of the durations of the fitted meters. These boundary offsets will be used to split timespans before they are fed to rhythm-makers for rhythm generation in order to ensure that no generated rhythm crosses any bar-lines:

```python
def get_meters_and_meter_boundaries(timespan_inventory, permitted_meters):
    offset_counter = metertools.OffsetCounter(timespan_inventory)
    fitted_meters = metertools.Meter.fit_meters_to_expr(
        expr=offset_counter,
        maximum_run_length=1,
        meters=permitted_meters,
    )
    meter_durations = [Duration(_).for _ in fitted_meters]
    meter_boundaries = mathtools.cumulative_sums(
        meter_durations,
        start=Offset(0),
    )
    return fitted_meters, meter_boundaries
```

### 3.9.2 Organizing timespans

Each of the performed timespans in the input timespan inventory was configured with a voice name, allowing that timespan to be associated with a voice context in the score hierarchy produced by the input score template. As demonstrated earlier, Abjad components can be indexed by context name regardless of the depth of the named context in the score hierarchy:

```python
>>> timespan = timespan_inventory[0]
>>> print(format(timespan))
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(1, 4),
    original_start_offset=durationtools.Offset(9, 2),
    original_stop_offset=durationtools.Offset(19, 4),
    voice_name='Voice 3',
)
```
In order to populate each voice in the score, the timespans in the timespan inventory need to be separated into voice-wise timespan inventories, each only containing timespans associated with the same voice. This can be accomplished by building a dictionary whose keys are voice names and whose values are timespan inventories:

```python
def get_all_voicewise_timespans(timespan_inventory):
    voicewise_timespans = {}
    for timespan in timespan_inventory:
        voice_name = timespan.voice_name
        if (voice_name not in voicewise_timespans):
            voicewise_timespans[voice_name] = timespantools.TimespanInventory()
            voicewise_timespans[voice_name].append(timespan)
    return voicewise_timespans
```

### 3.9.3 Populating voices

Creating *performed* music requires first splitting a contiguous inventory of performed timespans by meter boundaries, collecting the shards together into a new inventory and then calculating the durations of those timespans. The resulting sequence of durations, along with an input rhythm-maker and an optional seed value can be sent to another function for rhythm generation:

```python
def make_performed_music(rhythm_maker, timespans, meter_boundaries, seed=0):
    split_timespans = timespantools.TimespanInventory()
    for shard in timespans.split_at_offsets(meter_boundaries):
        split_timespans.extend(shard)
    durations = [_.duration for _ in split_timespans if _.duration]
    music = make_music(rhythm_maker, durations, seed=seed)
    return music
```

Similarly, *silent* music – rests between groups of performed timespans in each voice – can be created by constructing a timespan which outlines the gap between other performed timespans, splitting that timespan by meter boundaries, collecting the durations of the split shards, and finally passing those durations along with a rest-generating rhythm-maker to another function for rhythm generation. As demonstrated in subsection 3.5.1, rest-generating rhythm-makers can be created from note rhythm-makers which use output masks to silence all output divisions:
def make_silent_music(start_offset, stop_offset, meter_boundaries):
    silence_timespan = timespantools.Timespan(start_offset, stop_offset)
    shards = silence_timespan.split_at_offsets(meter_boundaries)
    durations = [_.duration for _ in shards if _.duration]
    mask = rhythmmakertools.silence_all()
    rhythm_maker = rhythmmakertools.NoteRhythmMaker(output_masks=[mask])
    music = make_music(rhythm_maker, durations)
    return music

Both silent and performed music generation relies on the same core rhythm generating function, `make_music()`. This function calls its input rhythm-maker with a sequence of durations and an optional seed value, which may be interpreted variably depending on what kind of rhythm-maker was used, but which generally results in the rotation of any sequence-like configured value in that rhythm-maker, such as talea. The product of this first operation is a sequence of component selections. Some massaging converts this selections into a sequence of trivially-prolated containers and non-trivially prolated tuplets which can then be wrapped inside a larger container representing a complete phrase:

def make_music(rhythm_maker, durations, seed=0):
    music = rhythm_maker(durations, rotation=seed)
    for i, division in enumerate(music):
        if len(division) == 1 and isinstance(division[0], scoretools.Tuplet):
            music[i] = division[0]
        else:
            music[i] = scoretools.Container(division)
    music = scoretools.Container(music)
    return music

### 3.9.4 Rewriting meters

After all phrases for a voice have been generated, their meters are rewritten. This process involves simultaneously iterating through both the divisions in each phrase along with pairs of meters and meter downbeat offsets – that is, the offset from the origin of the score where each meter begins. Unprolated division containers are rewritten according to the meter active when they begin. Because a division may start after a meter has begun it is necessary to calculate the difference between each unprolated division’s start offset and each meter’s start offset. This difference is passed to the `MutationAgent`’s `rewrite_meter()` method via the `initial_offset` keyword, allowing the division to be properly aligned against the desired meter. Prolated divisions – tuplets – are rewritten solely with respect to

---

13 More optimizations to this joint container / meter iteration are obviously possible. Some have even been implemented in Consort’s source. Such optimization include associating meters with their own timespans, and storing these meter-timespans in an optimized interval-tree data structure allowing rapid retrieval by both offset and timespan intersection. For the sake of pedagogical clarity, a more naïve approach is used here.
the duration of their contents, not to the prevailing meter. That is, a 5:4 tuplet in a ⁴⁄₈ measure is rewritten with a meter of ⁵⁄₈, not of ⁴⁄₈:

```python
def rewrite_meters(phrase, meters, meter_boundaries):
    pairs = list(zip(meters, meter_boundaries))
    for division in phrase:
        division_offset = inspect_(division).get_timespan().start_offset
        while 1 < len(pairs) and pairs[1][1] <= division_offset:
            pairs.pop(0)
        if isinstance(division, scoretools.Tuplet):
            contents_duration = division._contents_duration
            meter = metertools.Meter(contents_duration)
            mutate(division).rewrite_meter(meter, boundary_depth=1,)
        else:
            meter, meter_boundary = pairs[0]
            initial_offset = division_offset - meter_boundary
            mutate(division).rewrite_meter(meter, boundary_depth=1,
                                           initial_offset=initial_offset,)
```

3.9.5 Score post-processing

The final step in this score building process adds a “floating time signature context” to the score, filled with typographic spacer skips and time signature commands, one for each of the fitted meters. This context appears as a row of time signatures floating above the topmost staff in the score, allowing time signatures to be omitted from every staff in the score and thereby improving proportional notation spacing. The time signature context’s context_name property instructs LilyPond to look for a context definition with that name and apply any typographic overrides found there during the typesetting process. While LilyPond does not come packaged with a TimeSignatureContext definition, examples of what such a definition looks like can be found in the various stylesheet sections of the source code appendices to this document:

```python
def add_time_signature_context(score, meters):
    time_signatures = ... implied_time_signature for _ in meters]
    measures = scoretools.make_spacer_skip_measures(time_signatures)
    time_signature_context = scoretools.Context([measures],
                                               context_name='TimeSignatureContext',
                                               name='TimeSignatureContext',)
    score.insert(0, time_signature_context)
```
3.9.6 Examples

Calling the `build_score()` function generates a score as described above. In this example, the function’s arguments comprise a talea rhythm-maker, an inventory of meters, as well as the previously defined score template and timespan inventory:

```python
>>> permitted_meters = metertools.MeterInventory([
...    (2, 4), (4, 8), (3, 4), (6, 8), (7, 8), (4, 4),
...])
>>> performed_rhythm_maker = rhythmmakertools.TaleaRhythmMaker(
...    beam_specifier=rhythmmakertools.BeamSpecifier(
...        beam_each_division=True,
...        beam_divisions_together=True,
...    ),
...    extra_counts_per_division=(0, 1),
...    talea=rhythmmakertools.Talea([1, 2, 3, 4, 5], 16),
...)
>>> show(performed_rhythm_maker, divisions=divisions)
```

For clarity, the phrases and their internal divisions have been annotated via Consort’s `annotate` function. This function draws thick brackets underneath each staff, with the lower bracket indicating the overall phrase grouping and the inner brackets indicating divisions within each phrase. Entirely silent phrasing brackets are drawn with dashed rather than solid lines:

```python
>>> score = build_score(
...    performed_rhythm_maker=performed_rhythm_maker,
...    permitted_meters=permitted_meters,
...    score_template=score_template,
...    timespan_inventory=timespan_inventory,
...)
>>> consort.annotate(score)
>>> show(score)
```
Large-scale variations are possible simply by altering the arguments to the `build_score()` function. For example, the timespan inventory can be reflected around its axis and stretched from a duration of $19/4$ to $25/4$. This will change the overall duration and phrase density of the resulting score while maintaining the character of its surface rhythms:

```python
>>> multiplier = Duration(24, 4) / Duration(19, 4)
>>> timespan_inventory = timespan_inventory.reflect()
>>> timespan_inventory = timespan_inventory.stretch(multiplier)
>>> timespan_inventory = timespan_inventory.round_offsets(Duration(1, 8))
>>> show(timespan_inventory, key='voice_name')
```
>>> score = build_score(
...   performed_rhythm_maker=performed_rhythm_maker,
...   permitted_meters=permitted_meters,
...   score_template=score_template,
...   timespan_inventory=timespan_inventory,
...   )
>>> consort.annotate(score)
>>> show(score)
Many other variations and extensions to the basic score-building algorithm outlined above are possible. The input rhythm-maker could be varied or replaced by a composite rhythm-maker. The score generation process could be extended so that rhythm-makers could be stored on the performed timespans themselves, allowing each performed timespan to specify its local rhythmic language. Multiple timespan inventories could be combined together using the various logical operations to implement masking or fusing when timespans from one inventory overlap those from another. In fact, all of these extensions and variations are employed in Consort’s own score generation process, as described in chapter 4.
This page intentionally left blank.
Consort: a model of composition

 Consort, a Python library I have written as an extension to Abjad, models the process of composing notated musical scores in terms of a repeated cycle containing two distinct stages: specification and interpretation. During specification, the composer describes the structure of the score procedurally in terms of what musical materials – themselves modeled as bundles of procedures for generating notation – should appear at particular moments in time. Then, during interpretation, those procedural descriptions are gradually converted into notation, turning the abstract specification of structure into a concrete segment of score. What follows is a detailed analysis of the various algorithms and subroutines employed during Consort’s specification and interpretation stages.
4.1 Specification

Specification describes how out-of-time materials – both concrete and programmatic – should be deployed in-time in a segment of musical score as notation. Materials encompass abstractions – such as pitch sets or collections of performance technique indications – and procedures for producing, altering or embellishing music such as rhythm-makers or attachment-handlers. Score segments comprise any contiguous passage of music, demarcating an area of compositional concern. Consort treats scores as comprised of at least one segment, but potentially many more concatenated together. Any segment may of course contain arbitrarily complex inner structuring. Separation of scores into distinct segments acts then mainly as an aid for the composer, both by simplifying the complexity of the current specification under consideration, and by allowing the typesetting engine – LilyPond – to display more manageable amounts of notation than the full score, thus speeding up the cycle of specifying, interpreting and visualizing.

4.1.1 Segment-makers

Composers specify segments by creating and progressively configuring segment-makers, classes which conceptually mirror the rhythm- and timespan-makers described in chapter 3, but on a much larger scale. Such configuration parameters include tempo, permitted meters, desired duration, and score template. Score templates, introduced in subsection 2.8.4, are notation factories which build scores comprised of staff groups, staves, voices, clefs and instruments, as necessary to model the context hierarchy of a score to which no count-time components have yet been added. All segment-makers in a score project must use the same score template if they are to appear contiguously in the complete typeset score, as this mechanism makes use of LilyPond’s context concatenation behavior, demonstrated in subsection 5.2.3. The following defines a segment-maker with a desired duration of 9 seconds, only \( \frac{3}{4} \) meters permitted, a score template comprised of two rhythmic staves, and a tempo of 60 quarter-notes per minute:

```python
>>> segment_maker = consort.SegmentMaker(  
    ...     desired_duration_in_seconds=9,  
    ...     permitted_time_signatures=[(3, 4)],  
    ...     score_template=templatetools.GroupedRhythmicStavesScoreTemplate(  
    ...         staff_count=2,  
    ...         with_clefs=True,  
    ...     ),  
    ...     tempo=indicatortools.Tempo(((1, 4), 60),  
    ... )
```
This segment-maker can be illustrated via `show()`, like many other objects in Abjad. Illustration here invokes the segment-maker’s interpretation stage. The `verbose=False` flag prevents it from printing a considerable amount of diagnostic information:

```python
>>> show(segment_maker, verbose=False)
\( \frac{3}{4} \) \( \dot{\text{\texttt{\textbf{n}}} \text{=} 60} \\
```

By changing the tempo from quarter-equals-60 to quarter-equals-20, the overall notated duration of the segment shrinks by two thirds, but the duration in seconds remains the same. This mechanism allows segments to be planned relative one another in terms of their “actual” durations, even if their tempi differ:

```python
>>> slower_segment_maker = new(
...   segment_maker,
...   tempo=indicatorTools.Tempo((1, 4), 20),
... )
>>> show(slower_segment_maker, verbose=False)
\( \frac{3}{4} \) \( \dot{\text{\texttt{\textbf{n}}} \text{=} 20} \\
```

Most importantly, segment-makers may be configured with any number of music settings, which object-model both when and in which voices musical materials should be deployed.

### 4.1.2 Music settings

Music settings represent a layer of musical texture, in one or more voices, of arbitrary length. Music settings aggregate a timespan-maker, a target timespan and any number of music specifiers. The timespan-maker provides the overall phrasing and density structure, the optional timespan identifier defines in what portion of the current segment the timespan-maker’s texture should be spooled out, and the music specifiers define both in which voices the timespan texture should appear as well as how those timespans should ultimately be rendered as notation. The
order in which composers configure segment-makers with music settings defines each music setting’s layer. Layer
determines how overlapping events in a single voice will mask one another. The first setting defined receives layer
0, the second layer 1 and so forth, with each higher layer number indicating higher precedence or “foregroundness”.
The timespans created by music settings which are defined later during segment specification “hide” any timespans
created by those music settings defined earlier. Score materials, including music settings, music specifiers, timespan-
makers and any other class pertinent to score creation – potentially even other segment-makers, may be defined
from scratch in the same code module as the segment-maker currently being configured, templated from another
material, or simply imported into the segment definition’s namespace.

4.1.3 Music specifiers

Music specifiers bundle all of the information necessary for a segment-maker to generate the notational content
for a sequence of one or more divisions, grouped as a phrase. This information includes optional rhythm-maker
(section 3.5), grace-handler (subsection 4.4.2), pitch-handler (subsection 4.4.3) and attachment-handler (subsection
4.4.4) definitions – all classes which describe strategies for creating or modifying notation –, as well as a variety
of other properties including an optional minimum phrase duration – described further in subsection 4.3.4 –, and
seed (subsection 4.3.5).

Music specifiers can also be configured with a labels, a tuple of one or more arbitrary strings, identifying some
quality of those performed timespans. As demonstrated in subsection 3.4.3, composers configure dependent timespan-
makers in order to create timespans according to the disposition of performed timespans associated with specific
voices. Dependent timespan-makers can also be configured to select depended-upon timespans based on the labels
property of those performed timespans’ music specifiers, allowing an additional category for filtering. This helps
model creating a pedal voice mirroring not simply a pianist’s left- and right-hand events, but only those that actually
depress keys, ignoring guero events or other off-the-key percussive techniques for which no pedaling is desired.

Consider the following timespan inventory, populated with performed timespans associated with one of two
voices, and annotated with music specifiers which are either labeled or not:

```python
>>> unlabeled_musicSpecifier = consort.MusicSpecifier()
>>> labeled_musicSpecifier = consort.MusicSpecifier(labels=[’labeled’])
>>> timespan_inventory = timespantools.TimespanInventory([
...     consort.PerformedTimespan(
...         layer=1,
...         start_offset=0,
...         stop_offset=4,
```
A dependent timespan-maker, configured to select timespans associated with “Voice 1” and “Voice 2” produces dependent timespans in the manner illustrated in subsection 3.4.3:

```python
>>> dependent_timespan_maker = consort.DependentTimespanMaker(  ...
           ... include_inner_starts=True,
           ... voice_names=('Voice 1', 'Voice 2'),
           ... )
>>> result = dependent_timespan_maker(  ...
           ... layer=3,
           ... music_specifiers={'Voice 3': None},
           ... timespan_inventory=timespan_inventory[:],
           ... )
>>> show(result, key='voice_name')

Voice 1:
0
1
4
1
6
1
8
1
11
1
14
1
16
1
1 2 1 1
Voice 2:
2
1
7
1
10
1
25
2
15
1
16
1
1 2 1
Voice 3:
0
1
2
1
6
1
8
1
10
1
11
1
14
1
15
1
16
1
3 3 3 3 3 3 3

Reconfiguring the above dependent timespan-maker to additionally filter timespans whose music specifier is labeled “labeled” produces a more restricted output. Note that the unlabeled 6/4-8/4 timespan in “Voice 1” and the 10/4-25/2 timespan in “Voice 2” are ignored:

```python
>>> dependent_timespan_maker = new(  ...
           ... dependent_timespan_maker,
           ... labels=['labeled'],
           ... )
>>> result = dependent_timespan_maker(  ...
           ... layer=3,
           ... music_specifiers={'Voice 3': None},
           ... timespan_inventory=timespan_inventory[:],
           ... )
>>> show(result, key='voice_name')

Voice 1:
0
1
4
1
6
1
8
1
11
1
14
1
16
1
1 2 1 1
Voice 2:
2
1
7
1
10
1
25
2
15
1
16
1
1 2 1
Voice 3:
0
1
2
1
7
1
11
1
14
1
15
1
16
1
3 3 3 3 3

Music specifiers can be grouped into sequences called, unsurprisingly, music specifier sequences, allowing a music setting to specify that the timespans it creates for a particular voice should be annotated with multiple different music specifiers in a patterned way:
Recall from subsection 3.4.2 that talea timespan-makers can create contiguous groups of 1 or more timespan associated with a specific voice. Music specifier sequences can be used to annotate each timespan in a contiguous group with a different music specifier, or to simply choose a different music specifier for each contiguous group as a whole. This behavior is controlled via the music specifier’s application rate property:
Changing the application rate from the default value of “phrase” to “division” causes a different music specifier from the sequence to be annotated to each timespan, rather than each contiguous group of timespans:

```python
>>> music_specifier_sequence = new(
...    music_specifier_sequence,
...    application_rate='division',
...  )
>>> music_specifiers = {'Voice': music_specifier_sequence}
>>> timespan_inventory = timespan_maker(
...    music_specifiers=music_specifiers,
...    target_timespan=target_timespan,
...  )
>>> print(format(timespan_inventory))
timespantools.TimespanInventory([
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(1, 4),
        music_specifier='A',
        voice_name='Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(1, 4),
        stop_offset=durationtools.Offset(1, 2),
        music_specifier='B',
        voice_name='Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(1, 2),
        stop_offset=durationtools.Offset(3, 4),
        music_specifier='C',
        voice_name='Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(1, 1),
        stop_offset=durationtools.Offset(5, 4),
        music_specifier='B',
        voice_name='Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(5, 4),
        stop_offset=durationtools.Offset(3, 2),
```
Consort provides an additional variation of music specifier called a *composite music specifier*. Composite music specifiers allow for the definition of music for voices in tandem, such as the fingering- and bowing-voice in a split-hands string notation. When passed as part of a voice-name-to-music-specifier mapping to a timespan-maker, that timespan-maker will create timespans for the first voice of the composite music specifier and then create timespans for the second voice as though it had its own dependent timespan-maker solely for those timespans from the first voice. This behavior ensures a degree of synchronization between pairs of voices which should always appear at the same time in a score:

```python
>>> composite_music_specifier = consort.CompositeMusicSpecifier(
    ...     primary_musicSpecifier='one',
    ...     primary_voice_name='Viola 1 RH',
    ...     rotation_indices=(0, 1, -1),
    ...     secondary_voice_name='Viola 1 LH',
    ...     secondary_music_specifier=consort.MusicSpecifierSequence(
    ...         application_rate='phrase',
    ...         music_specifiers=['two', 'three', 'four'],
    ...     ),
    ... )
>>> music_specifiers = {'Viola 1': composite_music_specifier}
>>> timespan_inventory = timespan_maker(
    ...     music_specifiers=music_specifiers,
    ...     target_timespan=target_timespan,
    ... )
>>> show(timespan_inventory, key='voice_name')
```

Viola 1 LH:
```
0 1 4 1 2 3 1 1 5 4 3 2 7 4
```

Viola 1 RH:
```
0 1 4 1 2 3 1 1 5 4 3 2 7 4
```
4.2 Interpretation

At any point during specification, a segment-maker may be interpreted to produce an illustration. Score interpretation proceeds conceptually much like compilation in classical computing, where a compiler parses an instruction set written in some source language into an intermediate representation and then transforms that same intermediate representation into instructions executable on a target platform. In Consort’s interpretation stage, the compiler is the segment-maker itself, and the source instruction set its configuration – its tempo, permitted meters, music settings and so forth. Timespan inventories produced by each music setting’s timespan-maker, populated with timespans annotated with music specifiers perform the role of the intermediate representation. This intermediate representation acts as a maquette, blocking out where in the resulting score segment various materials should be deployed. The target of score interpretation is, unsurprisingly, a fully-fledge score aggregated from Abjad score components. Interpretation takes place in two broad stages – rhythmic interpretation, followed by non-rhythmic interpretation – with the first stage producing a score populated solely with rhythmic information, and the second stage applying grace notes, pitches, indicators, spanners and various typographic overrides to the previously-constructed rhythmic skeleton.

4.3 Rhythmic interpretation

Broadly speaking, rhythmic interpretation proceeds from coarse- to fine-grained. The segment-maker creates a maquette – a model of the locations of musical materials in the score – by calling each of its music-settings in turn to populate a timespan inventory. It then resolves overlap conflicts within that inventory, fits meters against the resolved inventory’s offsets, splits and prunes the contents of the inventory according to its fitted metrical structure, and finally converts the finished timespan maquette into an actual score. This process, like interpretation overall, can be roughly divided into work flows of maquette creation and music creation, although in practice the two flows are interleaved significantly as they actually influence one another. When creating the maquette, music settings with independent timespan-makers – those which do not depend on the contents of a previously created timespan inventory, specifically flooded and talea timespan-makers – are called in a first pass, and those with dependent timespan-makers in a second. These two passes only differ significantly in that meters are fitted against the segment’s timespan maquette during the independent timespan-maker pass, but not during the dependent.
4.3.1 Populating the maquette

To populate the maquette, the segment-maker calls each of its music settings to produce timespans according to their configured timespan-makers, *timespan-identifiers* – optional specifications of which portion of the segment’s overall timespan to operate within – and voice-associated music specifiers. Timespan identifiers may include timespans, inventories of timespans, or even expressions callable against the segment-makers own timespan which evaluate to an inventory of timespans.

Music settings exist without any reference to a segment-maker, its desired duration – and therefore desired timespan –, or its score template. In order to know which target timespan or timespans a music setting’s timespan-maker should operate within – in the case of procedural timespan identifiers such as *ratio-parts expressions* which must be called against a preexisting timespan in order to determine what part or parts of that timespan to use – the music setting must resolve its timespan identifier against the segment’s desired duration. Target timespan resolution must also take into account offset quantization, as the target timespans resulting from the evaluation of a ratio-parts expression may not align against a power-of-two-denominator offset grid such as $\frac{1}{8}$, $\frac{1}{16}$ or $\frac{1}{32}$. Because timespan-makers produce their output relative to the start-offset of their target timespan, a misaligned target timespan – starting at an offset like $\frac{1}{3}$ or $\frac{15}{7}$ rather than $\frac{1}{4}$ or 0 – will cause all generated timespans to be misaligned.

Music settings associate their music specifiers with strings containing voice-name abbreviations. These abbreviations are always underscore-delimited strings such as *violin_1* or *piano_lh* – necessitated by Python’s keyword argument syntax so that they can be used as keys during class instantiation – which represent voices in a score, without having established a concrete reference to those voice contexts. In order to match its music specifiers against actual voice contexts in a score, the music setting must resolve its voice-name abbreviations against a score template, looking up each abbreviation on the template and returning the real name of the associated context. This lookup process allows music settings to construct well-formed voice-name-to-music-specifier mappings, implemented as ordered dictionaries and ordered by the actual *score index* – effectively, the vertical location – of each looked-up context in the segment-maker’s under-construction score. As demonstrated in section 3.4, timespan-makers require these mappings to produce their output. Additionally, voice-name resolution guarantees that the values in the resolved voice-name-to-music-specifier mapping are always either a *MusicSpecifierSequence* or *CompositeMusicSpecifier* instance via coercion, where any composite music-specifier’s primary and secondary music specifiers are themselves coerced into music specifier sequences. This coercion ensures that all arguments to the music setting’s
timespan-maker are in a well-formed and predictable state.\textsuperscript{1}

Consider the following string quartet score template, which will be used with the above music setting. Note the names of the various contexts defined in it, made visible when formatted as LilyPond syntax, where each context name is given by a quoted string on the lines beginning with \context. The score contains a time signature context, and four staff groups, one for each instrument in the quartet. These staff groups then contain two staves, for the left and right hands of each performer, with each staff containing a single voice:

\begin{verbatim}
>>> score_template = consort.StringQuartetScoreTemplate()
>>> score = score_template()
>>> print(format(score))
\context Score = "String Quartet Score" <<
  \tag #'time
  \context TimeSignatureContext = "Time Signature Context" { }
  \tag #'violin-1
  \context StringPerformerGroup = "Violin 1 Performer Group" \with {
    instrumentName = \markup {
      \center-ln \
      #10
      "Violin 1"
    }
    shortInstrumentName = \markup {
      \center-ln \
      #10
      "Vln. 1"
    }
  }
  \context BowingStaff = "Violin 1 Bowing Staff" {
    \clef "percussion"
    \context Voice = "Violin 1 Bowing Voice" {
    }
  }
  \context FingeringStaff = "Violin 1 Fingering Staff" {
    \clef "treble"
    \context Voice = "Violin 1 Fingering Voice" {
    }
  }
>>>
  \tag #'violin-2
  \context StringPerformerGroup = "Violin 2 Performer Group" \with {
    instrumentName = \markup {
      \center-ln \
    }
  }
\end{verbatim}

\textsuperscript{1}Timespan-makers actually delegate the creation of performed and silent timespans to music specifier sequences. While not demonstrated explicitly in section 3.4, this delegation allows timespan-makers to use both music specifier sequences and composite music specifiers interchangeably, with the former creating timespans associated with one voice and the later with two. When the values of a timespan-maker's input voice-name-to-music-specifier mapping are neither music specifier sequences nor composite music specifiers – as was the case in all of the examples in section 3.4 – they implicitly coerce those values into music specifier sequences.
Consort’s score templates provide abbreviation-to-voice-name mappings via their `context_name_abbreviations` property. Likewise, they provide mappings for use with composite music specifiers which map “parent” context abbreviations to their relevant child abbreviation pairs:

```python
for abbr, context_name in score_template.context_name_abbreviations.items():
    print(abbr, context_name)

{('viola_rh', 'Viola Bowing Voice')
 ('cello_rh', 'Cello Bowing Voice')
 ('violin_1_lh', 'Violin 1 Fingering Voice')
 ('violin_2_lh', 'Violin 2 Fingering Voice')
 ('violin_2_rh', 'Violin 2 Bowing Voice')
 ('cello', 'Cello Performer Group')
 ('viola', 'Viola Performer Group')
 ('violin_1_rh', 'Violin 1 Bowing Voice')
 ('viola_lh', 'Viola Fingering Voice')
 ('violin_2', 'Violin 2 Performer Group')
 ('cello_lh', 'Cello Fingering Voice')
 ('violin_1', 'Violin 1 Performer Group')

for parent, child_pair in score_template.composite_context_pairs.items():
    print(parent, child_pair)

{('violin_2', ('violin_2_rh', 'violin_2_lh'))
 ('cello', ('cello_rh', 'cello_lh'))
 ('viola', ('viola_rh', 'viola_lh'))
 ('violin_1', ('violin_1_rh', 'violin_1_lh'))
```

Consider the following music setting, which includes a timespan-maker definition, a ratio-parts expression as its timespan identifier, and three music specifiers, one of which is a composite music specifier. String values for the music specifiers are used simply for pedagogical reasons:
Resolving the above music specifier against Consort's string quartet score template dereferences the correct context names, allowing them to be indexed into any score created by that score template. Note that not only are the two non-composite music specifiers now associated with voices in the score, but they have been recreated as music specifier sequences. Likewise, the composite music specifier has been reconfigured such that it knows the specific names of the two voices associated with its primary and secondary music specifiers, themselves recreated as music specifier sequences:

```python
>>> result = music_setting.resolve_music_specifiers(score_template)
>>> for context_name, resolved_musicSpecifier in result.items():
...     print('CONTEXT:', context_name)
...     print(format(resolved_musicSpecifier))
... ('CONTEXT:', 'Violin 2 Fingering Voice')
consort.tools.MusicSpecifierSequence(
    music_specifiers=datastructuretools.CyclicTuple(["A"],)
)
('CONTEXT:', 'Viola Fingering Voice')
consort.tools.MusicSpecifierSequence(
    music_specifiers=datastructuretools.CyclicTuple(["B", "C", "D"],)
)
('CONTEXT:', 'Cello Performer Group')
consort.tools.CompositeMusicSpecifier(
    primary_music_specifier=consort.tools.MusicSpecifierSequence(
        music_specifiers=datastructuretools.CyclicTuple(["one"],)
    ),
    primary_voice_name='Cello Bowing Voice',
)```
secondary_music_specifier=consort.tools.MusicSpecifierSequence(
    music_specifiers=datastructuretools.CyclicTuple(
        ['two', 'three', 'four'],
    ),
    secondary_voice_name='Cello Fingering Voice',
)

Resolving a music setting’s timespan identifier against the segment-maker’s segment timespan results in one or more target timespans which can be used as argument to the music setting’s timespan-maker. We can simulate resolution against an actual timespan-maker’s timespan by simply creating a “segment timespan” from scratch:

```python
>>> segment_timespan = timespantools.Timespan(0, 8)
>>> show(segment_timespan)
```

```plaintext
0 1 2 3 4 5 6 7 8
```

```python
>>> target_timespans = music_setting.resolve_target_timespans(segment_timespan)
>>> show(target_timespans, range_=(0, 8))
```

```plaintext
0 1 2 3 4 5 6 7 8
```

Note that the start and stop offsets of the target timespans resolved above do not all align at offsets with power-of-two denominators, such as $\frac{1}{2}$, $\frac{1}{4}$ or $\frac{1}{8}$. By specifying a timespan quantization, the target timespans generated during resolution can be quantized to a grid:

```python
>>> target_timespans = music_setting.resolve_target_timespans(
...    segment_timespan,
...    timespan_quantization=Duration(1, 16),
...)
>>> show(target_timespans, range_=(0, 8))
```

```plaintext
0 1 2 3 4 5 6 7 8
```

Additionally, a subtractive mask can be applied to the target timespans:

```python
>>> music_setting = new(
...    music_setting,
...    timespan_identifier__mask_timespan=timespantools.Timespan(
...        start_offset=(1, 2),
...        stop_offset=7,
...    ),
...)
>>> target_timespans = music_setting.resolve_target_timespans(segment_timespan)
>>> show(target_timespans, range_=(0, 8))
```

```plaintext
0 1 2 3 4 5 6 7 8
```
Once resolved, each music setting can call its timespan-maker to create timespans with the appropriate voice-name-to-music-specifier mapping, target timespans and layer, adding the contents of the resulting inventory to the growing maquette of performed and silent timespans produced by previous music settings. The populating process repeats until no more music settings remain.

4.3.2 Resolving cascading overlap

One of the driving motivations behind Consort is the ability to create musical textures consisting of multiple overlapping layers, each created by an independent maker and each with different materials from the other layers, allowing multiple materials of various provenances to appear in the same instrumental voice. Still, because acoustic instruments cannot simply “create” arbitrary numbers of voices like a synthesizer might, any overlap in material allocated for a given voice needs to be resolved. Consort handles resolution of overlap via tournament, choosing only one material from a collection of overlapping candidates.

Consider the following three timespan inventories, created with three different timespan-makers and music-specifier mappings. The first inventory is created via a flooded timespan-maker, filling the entirety of a target timespan of $\frac{0}{4}$ to $\frac{19}{4}$ in voices “Voice 2” and “Voice 3”. This inventory behaves like a constant “background layer” for those two voices:

```python
>>> layer_1_timespan_maker = consort.FloodedTimespanMaker()
>>> layer_1_target_timespan = timespantools.Timespan(0, (19, 4))
>>> layer_1_music_specifiers = collections.OrderedDict[
...     ('Voice 2', None),
...     ('Voice 3', None),
... ]
>>> layer_1 = layer_1_timespan_maker(
...     layer=1,
...     music_specifiers=layer_1_music_specifiers,
...     target_timespan=layer_1_target_timespan,
... )
>>> show(layer_1, key='voice_name', range_=(0, (21, 4)))

Voice 2:
\begin{align*}
\frac{1}{2} & \frac{19}{2} \\
\frac{10}{2} & \frac{1}{2}
\end{align*}

Voice 3:
\begin{align*}
\frac{1}{2} & \frac{19}{2} \\
\frac{10}{2} & \frac{1}{2}
\end{align*}
```

The second timespan inventory is created by a talea timespan-maker. This inventory covers all four voices – “Voice 1”, “Voice 2”, “Voice 3” and “Voice 4” – with a texture of evenly distributed phrases and silences. However, un-
like the first timespan inventory, this texture only spans the target timespan of $3/4$ to $19/4$, guaranteeing that the background layer for “Voice 2” and “Voice 3” is untouched in the span of $0/4$ to $3/4$:

```python
>>> layer_2_timespan_maker = consort.TaleaTimespanMaker(
...     initial_silence_talea=rhythmmakertools.Talea(
...         counts=(0, 1, 3),
...         denominator=8,
...     ),
...     playing_groupings=(1, 2),
...     playing_talea=rhythmmakertools.Talea(
...         counts=(1, 2, 3, 4),
...         denominator=4,
...     ),
...     silence_talea=rhythmmakertools.Talea(
...         counts=(5, 3, 1),
...         denominator=8,
...     ),
... )
>>> layer_2_target_timespan = timespantools.Timespan((3, 4), (19, 4))
>>> layer_2_music_specifiers = collections.OrderedDict(
...     ('Voice 1', None),
...     ('Voice 2', None),
...     ('Voice 3', None),
...     ('Voice 4', None),
... )
>>> layer_2 = layer_2_timespan_maker(
...     layer=2,
...     music_specifiers=layer_2_music_specifiers,
...     target_timespan=layer_2_target_timespan,
... )
>>> show(layer_2, key='voice_name', range_=(0, (21, 4)))

Voice 1:
3 4 1 1 13 8 17 8 23 8 13 4 17 4 35 8 37 8 2 2 2 2 2

Voice 2:
7 8 11 17 11 15 13 15 8 2 2 2 2

Voice 3:
9 8 13 19 5 7 2 2 3 3 4 4 4

Voice 4:
3 4 5 4 2 1 19 8 27 8 7 2 15 4 17 4 2 2 2 2 2

The third timespan inventory consists of groups of near-simultaneous attacks in three voices – “Voice 1”, “Voice 3” and “Voice 4”. This inventory’s talea timespan-maker has padded $1/4$-duration silences around the beginning and end of each group, guaranteeing that any performed timespans with layers lower than 3 will be masked not only by this layer’s performed timespans, but by its silent timespans as well. Additionally, the third timespan inventory was created with a target timespan of $6/4$ to $21/4$. While – due to the complexities of the talea timespan-maker’s patterns
– the generated timespan texture may not extend all of the way to its target timespan’s stop offset at 21\(\frac{1}{4}\), it will certainly not contain any performed or silent timespans earlier than 5\(\frac{3}{4}\) – the target timespan’s start offset minus the timespan-maker’s padding –, leaving lower layers untouched from 0\(\frac{1}{4}\) to 5\(\frac{3}{4}\):

```python
>>> layer_3_timespan_maker = consort.TaleaTimespanMaker(
    ...     initial_silence_talea=rhythmmakertools.Talea(
    ...         counts=(0, 0, 0, 1),
    ...         denominator=8,
    ...     ),
    ...     playing=Duration(1, 4),
    ...     playing_talea=rhythmmakertools.Talea(
    ...         counts=(2, 3, 4),
    ...         denominator=8,
    ...     ),
    ...     silence_talea=rhythmmakertools.Talea(
    ...         counts=(6,),
    ...         denominator=4,
    ...     ),
    ...     synchronize_step=True,
    ... )
>>> layer_3_target_timespan = timespantools.Timespan((6, 4), (21, 4))
>>> layer_3_music_specifiers = collections.OrderedDict(
    ...     ('Voice 1', None),
    ...     ('Voice 3', None),
    ...     ('Voice 4', None),
    ... )
>>> layer_3 = layer_3_timespan_maker(
    ...     layer=3,
    ...     music_specifiers=layer_3_music_specifiers,
    ...     target_timespan=layer_3_target_timespan,
    ... )
>>> show(layer_3, key='voice_name', range_=0, (21, 4))
```

Recall from section 3.4 that timespan-makers can modify a timespan inventory in-place when called, rather than generating a new one from scratch. The following code – greatly simplified from Consort’s `SegmentMaker.populate_multiplexed_maquette()` method – demonstrates the process of calling multiple timespan-makers with their corresponding target timespans and voice-name-to-music-specifier mappings to progressively populate a single timespan inventory in-place. Note the use of two Python built-in iterators `zip()` and `enumerate()`. The `zip()` iterator iterates over the iterables with which it was instantiated, yielding the first item of each of its iterables as a tuple, then the
second of each of its iterables, then the third, and so forth. The `enumerate()` iterator yields each item of its input iterable paired with that item’s index, filling the role in Python for the verbose for loop loop idiom found in many C-like languages, such as Java or Javascript: `for(int x = 10; x < 20; x = x + 1) { ... }.

```python
>>> timespan_makers = (
...    layer_1_timespan_maker,
...    layer_2_timespan_maker,
...    layer_3_timespan_maker,
... )
>>> music_specifiers = (
...    layer_1_music_specifiers,
...    layer_2_music_specifiers,
...    layer_3_music_specifiers,
... )
>>> target_timespans = (
...    layer_1_target_timespan,
...    layer_2_target_timespan,
...    layer_3_target_timespan,
... )
>>> triples = zip(timespan_makers, music_specifiers, target_timespans)
>>> timespan_inventory = timespantools.TimespanInventory()
>>> for layer, triple in enumerate(triples, 1):
...    timespan_inventory = triple[0](
...        layer=layer,
...        music_specifiers=triple[1],
...        target_timespan=triple[2],
...        timespan_inventory=timespan_inventory,
...    )
...```

Recall that the result of this process is still a timespan inventory, just as described in section 3.2. While this document has taken pains to clarify the internal structure of timespan-maker-generated timespan inventories by sorting and displaying voice-names in their illustrations – just as in the above three timespan inventory illustrations – that behavior derives from the timespan inventory’s illustration protocol implementation, and does not reflect their actual, flat structure:

```python
>>> show(timespan_inventory, range_=\((0, (21, 4)))
```

![Diagram of timespan inventory](image)
Just because timespan inventories are often displayed with voice-names, and contain performed timespans with voice-name attributes, does not mean that they are automatically or internally structured that way. Such sorting requires an additional pass – demultiplexing. Visualizing the inventory, exploded by voice-name, then sorted by layer, shows the overlap in each voice:

```python
>>> show(timespan_inventory, key='voice_name', sortkey='layer', range_=(0, (21, 4)))
```

In order to resolve cascading overlap, the segment-maker must first demultiplex the performed and silent timespans in the still-multiplexed maquette into separate timespan inventories by their voice-name attributes. The segment-maker further separates each demultiplexed-by-voice-name timespan inventory into multiple timespan inventories according to their contents' layer attributes, with the lowest-layered inventory first, and the highest-layered inventory last. This results in one inventory per-voice, per-timespan-maker from the maquette population process.² With the maquette fully demultiplexed, the segment-maker can proceed through each layer in each voice, from lowest to highest. It progressively subtracts the timespans in each higher inventory from the lowest inventory, effectively cutting out holes outlining the shapes of that higher inventory’s timespans. The segment-maker then adds that higher inventory’s timespans into the lowest-layered inventory. This process masks lower-layered timespans with higher ones. The process repeats until no more timespan inventories remain for that voice, then moves onto the inventories for the next voice. The resolved, demultiplexed results are finally collected into a timespan inventory mapping, associating voice names with resolved timespan inventories, and returned.

²Why not keep all timespan-maker output separated from the very beginning? Working with a single multiplexed timespan inventory for much of the rhythmic interpretation process simplifies many of the procedures used therein, such as dependent timespan-maker evaluation, splitting, consolidation, etc. Compared to many of the later rhythmic operations, such as meter rewriting, multiplexing and demultiplexing timespan inventories are computationally trivial.
After resolution, no overlap remains in the timespans for any voice. Note too that no silent timespans – like those created as padding in the third timespan-maker above – remain either. Silent timespans act solely as a means of “holding space” for a layer, masking but not replacing timespans in lower layers:

```python
>>> show(demultiplexed_maquette, range_=0, (21, 4))
```

Voice 1:

```
3 1 3 7 2 17 23 13 20 31 35 37
```

Voice 2:

```
7 11 17 11 15 33 35 19
```

Voice 3:

```
5 3 15 17 5 7 15 33 35 19
```

Voice 4:

```
5 3 2 19 27 15 17
```

Unlike many of the timespan-handling functions demonstrated in this chapter as well as in chapter 3, `resolve_maquette()` returns a `TimespanInventoryMapping` rather than a `TimespanInventory`. The timespan inventory mapping already explicitly uses voice-names as keys, obviating the need for a `key='voice_name'` keyword argument pair in the call to `show()`.

### 4.3.3 Finding meters, revisited

Consort’s segment-maker implements a variation on the meter-fitting algorithm described in section 3.8. Each segment-maker may be configured with an inventory of permitted meters, as well as maximum meter run length, in order to drive the meter fitting algorithm. When counting offsets, segment-makers include the offsets found on the performed timespans in their maquette but discard those from silent timespans, removing any influence from timespans created solely for silencing other timespans. The start offset of each performed timespan is weighed twice as much as their stop offset. This imbalance helps emphasize simultaneous phrase starts across different voices. Additionally, segment-maker’s weight their own desired stop offset at a much higher value than any count derived from the offsets in their maquette. This attempts to influence the meter fitting process into selecting a series of meters which end as close to their desired stop offset as possible. After fitting meters, the segment-maker caches both the fitted meters and their boundaries as properties on its instance, affording later retrieval by other subroutines.
4.3.4 Splitting, pruning & consolidation

Once meters have been fitted against the resolved maquette, the timespans in the maquette must be split at the measure boundaries outlined by those meters. Splitting guarantees that no timespans cross any bar-lines and that therefore no containers generated by those timespans when notating them as score components cross any bar-lines either. While LilyPond can typeset bar-line crossing notes, chords and even tuplets, the scores I have composed via Consort do not currently make use of such constructions. As described in subsection 3.1.3, operations on timespans which change offsets – generating new timespans in the process, rather than modifying the operated-upon timespan in-place – preserve their unmodified properties via templating. Splitting is no exception, and those timespans split maintain their music specifiers, layer identifiers and voice-names:

```python
>>> performed_timespan = consort.PerformedTimespan(  ...
    ...     layer=3,
    ...     start_offset=(1, 2),
    ...     stop_offset=(13, 8),
    ...     voice_name='Percussion Voice',
    ... )
>>> shards = performed_timespan.split_at_offset((9, 16))
>>> print(format(shards))
timespantools.TimespanInventory(
    [  
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(1, 2),
            stop_offset=durationtools.Offset(13, 8),
            layer=3,
            voice_name='Percussion Voice',
        ),
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(9, 16),
            stop_offset=durationtools.Offset(13, 8),
            layer=3,
            voice_name='Percussion Voice',
        ),
    ]
)
```

After splitting, the segment-maker prunes timespans considered either too short or malformed. Performed timespans may be configured with a `minimum_duration` property. Timespan-makers may set this property on timespans they create when they are themselves configured with a `TimespanSpecifier`. Any performed timespan whose actual duration is less than its minimum duration – if it has been configured with a minimum duration – will be removed from the maquette. Likewise any timespan with a duration of 0 – therefore malformed – will also be removed.
While the latter pruning guarantees correctness of the maquette – malformed timespans cannot be rendered as notation at all, and may cause other problems when partitioning due to ambiguities in their start / stop offset semantics –, the former allows for a kind of compositional control over the maquette. When notated with certain rhythm-makers, overly short divisions – especially those shorter than $\frac{1}{8}$-duration – may give undesirable results.

Note that silent timespans have no configurable minimum duration. Their $\text{minimum\_duration}$ always returns 0. They maintain this dummy property so that the segment-maker’s timespan-pruning algorithms can treat silent and performed timespans identically.

Next, Consort’s segment-maker consolidates contiguous performed timespans with identical music specifiers, caching the durations of the consolidated timespans in a new timespan’s $\text{divisions}$ property. Each new consolidated timespan outlines the start and stop offset of its consolidated group:

```python
>>> timespans = timespantools.TimespanInventory(
    ... consort.PerformedTimespan(
    ...     start_offset=0,
    ...     stop_offset=10,
    ...     music_specifier='foo',
    ... ),
    ... consort.PerformedTimespan(
    ...     start_offset=10,
    ...     stop_offset=20,
    ...     music_specifier='foo',
    ... ),
    ... consort.PerformedTimespan(
    ...     start_offset=20,
    ...     stop_offset=25,
    ...     music_specifier='bar',
    ... ),
    ... consort.PerformedTimespan(
    ...     start_offset=40,
    ...     stop_offset=50,
    ...     music_specifier='bar',
    ... ),
    ... consort.PerformedTimespan(
    ...     start_offset=50,
    ...     stop_offset=58,
    ...     music_specifier='bar',
    ... ),
    ... )
>>> show(timespans)
```

```
0   10   20   25   40   50   58
```

```python
>>> consolidated_timespans = consort.SegmentMaker.consolidate_timespans(timespans)
>>> show(consolidated_timespans)
```
Consolidation transforms performed timespans from free-floating cells in the maquette into components of larger phrases. The cached divisions also prepare these consolidated timespans for inscription by defining the correct input for a rhythm-maker: a sequence of divisions.

```python
>>> print(format(consolidated_timespans))
timespantools.TimespanInventory(
    [
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(20, 1),
            divisions=(
                durationtools.Duration(10, 1),
                durationtools.Duration(10, 1),
            ),
            music_specifier='foo',
        ),
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(20, 1),
            stop_offset=durationtools.Offset(25, 1),
            divisions=(
                durationtools.Duration(5, 1),
            ),
            music_specifier='bar',
        ),
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(40, 1),
            stop_offset=durationtools.Offset(58, 1),
            divisions=(
                durationtools.Duration(10, 1),
                durationtools.Duration(8, 1),
            ),
            music_specifier='bar',
        ),
    ]
)
```

If the music specifier of the consolidated timespan was configured with a minimum phrase duration, and the consolidated timespan falls under that threshold, it too is discarded.

### 4.3.5 Inscription

Inscription describes the process of generating music from a performed timespan’s divisions and rhythm-maker and inscribing the timespan with the result. Consort’s segment-maker performs inscription by iterating over the timespans for each voice in the demultiplexed, consolidated maquette, in score order. For each performed timespan encountered, the segment-maker retrieves that performed timespan’s music specifier, and increments that music
specifiers count in a counter. This allows each music specifier to maintain a seed value while inscribing each performed timespan, even in fragmentary textures, and to produce continuously varied results from each successive rhythm-maker belonging to the same music specifier. Recall from subsection 3.9.3 that rhythm-makers can be called not only with a list of divisions, but also a seed value, rotating the rhythm-makers sequence-like properties when creating its rhythmic output. All of the handlers discussed in section 4.4 employ similar – or even more complex – techniques for maintaining state across different phrases sharing the same music specifier. The segment-maker also retrieves the performed timespan’s division list – created during consolidation, as described in subsection 4.3.4 – and its rhythm-maker. Rhythm-maker retrieval, like seed retrieval, is non-trivial. A performed timespan’s music specifier may not have rhythm-maker defined, or that performed timespan may not even have a music specifier defined. If a performed timespan has a rhythm-maker defined on its music specifier, the segment-maker retrieves that. If the timespan has a music specifier, but no defined rhythm-maker, the segment-maker constructs a note rhythm-maker which ties all of its divisions together. If the timespan has no music specifier defined at all, the segment-maker returns a fully-masked note rhythm-maker.

With the performed timespan’s seed, rhythm-maker and division list ready, the segment-maker creates the performed timespan’s music. This proceeds almost identically to the `make_music()` function described in subsection 3.9.3. Consort’s segment-maker makes one additional adjustment on top of that algorithm, replacing trivially-prolated tuplets – tuplets with ratios of 1:1 – with unprolated containers. Next, the segment-maker performs rest consolidation on the newly-generated phrase, grouping all of the phrase’s unprolated rests – those not appearing in tuplets – into their own containers, and leaving all other components – all notes and chords, and any rests found within a tuplet – in their original division within the phrase. Rest consolidation allows the segment-maker to not only regroup the contents of a phrase into silent and non-silent segments, but to actually split the performed timespan itself, creating larger gaps within the maquette, and improving the chances for notating full-bar rests when finally filling in silences between phrases.

Consider the following phrase-like container – annotated to show its internal division structure –, containing divisions in various configurations – with rests at the beginning, at the end, with prolated rests, no rests at all, and so forth:

```plaintext
>>> parseable = r'''...
    \new Voice {
        ...
        { \time 4/4 r4 c4 }
        ...
        { \times 2/3 { c4 r8 } r4 }
```
A clear comparison can be made by duplicating the original phrase, consolidating its rests, annotating it, and grouping both the original unconsolidated phrase and the consolidated into a staff group, with the original above and the altered below:
Note above how after rest consolidation the rests in the lower staff have been grouped together into their own divisions – annotated with dashed lines – while all other components – including rests within tuplets – remain in their original divisions. Now consider the following simple rhythm-maker which, when passed a division sequence of seven $\frac{1}{4}$-durations, generates a rhythm consisting of two groups of $\frac{1}{4}$ notes delimited by rests:

```python
>>> rhythm_maker = rhythmmakertools.NoteRhythmMaker(
    ...     output_masks=[
    ...         rhythmmakertools.BooleanPattern(
    ...             indices=[0],
    ...             period=3,
    ...         ),
    ...     ],
    ... )
>>> divisions = [Duration(1, 4)] * 7
>>> show(rhythm_maker, divisions=divisions)
```

The above rhythm-maker can be treated as the rhythm-maker for a music specifier annotating a performed timespan. Likewise, the divisions used above can be used as the division sequence for this same performed timespan. If this timespan’s rhythm-maker would be called with its division list, the same rhythm as above would result:

```python
>>> timespan = consort.PerformedTimespan(
    ...     divisions=divisions,
    ...     start_offset=0,
    ...     stop_offset=(7, 4),
    ...     musicSpecifier=consort.MusicSpecifier(
    ...         rhythm_maker=rhythm_maker,
    ...     ),
    ... )
>>> show(timespan)
```

The timespan can then be inscribed by calling the segment-maker’s `inscribe_timespan()` method:

```python
>>> inscribed_timespans = consort.SegmentMaker.inscribe_timespan(timespan)
>>> show(inscribed_timespans, range_=(0, (7, 4)))
```

```python
>>> print(format(inscribed_timespans))
timespantools.TimespanInventory(
```
After inscription – including rest consolidation –, the original performed timespan has been split into two new performed timespans, whose start and stop offsets outline only those portions of the generated rhythm which do not contain rests. Their music attributes have likewise been populated with only the non-silent portions of that rhythm. During inscription, the segment-maker replaces each performed timespan in the maquette against which
inscription was called with the result of that inscription process. Additionally, the segment-maker performs some simple post-processing on each inscribed timespan’s music, attaching both a beam spanner to the leaves of the phrase, as well as the music specifier used to specify the phrase’s music to the phrase itself:

```python
>>> music = inscribed_timespans[0].music
>>> indicator = inspect_(music) get_indicator(consort.MusicSpecifier)
>>> print(format(indicator))
consort.tools.MusicSpecifier(
    rhythm_maker=rhythmmakertools.NoteRhythmMaker(
        output_masks=rhythmmakertools.BooleanPatternInventory(
            
            rhythmmakertools.BooleanPattern(
                indices=(0,),
                period=3,
            ),
        ),
    ),
)
```

Attaching the music specifier directly to the performed timespan’s phrase container allows each leaf in the segment-maker’s score to locate the music specifier which specified, without needing to resort to referencing the maquette.

### 4.3.6 Meter pruning

After the timespan pruning outlined in subsection 4.3.4, and the possibility of gaps introduced due to rest consolidation as outlined in subsection 4.3.5, the overall stop offset of the maquette – not the stop offset derived from the segment-maker’s desired duration – may have shifted earlier. Depending on the degree of shift, timespans in the maquette may no longer occur during one or more of the implicit timespans of the previously fitted meters. Segment-makers may be configured to discard these silences via their discard_final_silence property, progressively removing meters from the end of the list of fitted meters until one overlaps at least one performed timespan in the maquette.

### 4.3.7 Populating dependent timespans

The previous few passages, from subsection 4.3.1 through subsection 4.3.6, describe the process of populating a segment-maker’s timespan maquette with the products of its independent music settings – those music settings whose timespan makers are independent, notably flooded and talea timespan-makers. With the maquette partially populated, those music settings with dependent timespan-makers – timespan-makers which generate timespans based on the contents of a preexisting timespan inventory – may finally be called to provide their contributions.
Dependent population proceeds almost identically to independent population with a few notable differences. For one, dependent population dispenses with meter finding entirely. Consort treats meter as entirely dependent upon independent timespans, as dependent timespans – in practice – are generally used for keyboard pedaling voices, and should therefore have little bearing on the overall metrical structure. And while maquette resolving, as described in subsection 4.3.2, results in a demultiplexed timespan inventory mapping – a dictionary of voice-names to timespan inventories –, independent timespan population completes by re-multiplexing that mapping – effectively flattening – back into a single timespan inventory. This flattening prepares the correct input for dependent timespan population, as dependent timespans require a single pre-populated timespan inventory as input. As there are no more passes of timespans to add after dependent timespan population – discounting the population of silent timespans, as described in subsection 4.3.8 – dependent timespan population completes with its maquette still demultiplexed.

4.3.8 Populating silent timespans

With the maquette finally populated by inscribed performed timespans, properly split, resolved and consolidated, the segment-maker can fill in the remaining gaps between phrases in each voice. This is accomplished by creating rest-inscribed timespans for each gap. As there are no more layers to add to the maquette, the segment-maker can populate, split and inscribe timespans for each of these gaps in a single pass. For each voice in the segment-maker’s still-empty score, the segment-maker retrieves all timespans – if any – associated with that voice and subtracts each of them in turn from a single silent timespan the length of the entire segment, as determined by the first and last meter boundaries. Any remaining shards from that segment-length silent timespan represent gaps between phrases in that voice. If the maquette contains no performed timespans associated with that voice, the segment-length silent timespan remains unaltered. The segment-maker then splits the remaining silent timespan shards at every intersecting meter boundary, as described in subsection 4.3.4, guaranteeing that the resulting silent timespans do not cross bar-lines. Once split, the segment-maker partitions the silent timespans, and iterates over the partitioned groups. For each group of contiguous silent timespans, it generates a phrase of music containing only rests using a fully-masked note rhythm-maker – as described in subsection 4.3.5 –, attaches a dummy music-specifier to the phrase and instantiates a performed timespan annotated with that phrase, adding it to the current voice’s timespan inventory in the demultiplexed maquette.

Recall the demultiplexed maquette created earlier in subsection 4.3.2:

```python
>>> show(demultiplexed_maquette, range_=(0, 6))
```
The above assumes a segment duration of 6 instead of the maquette’s initial duration of $\frac{21}{4}$, and therefore pads out the end of each timespan inventory to that stop offset with silence.

### 4.3.9 Rewriting meters, revisited

Once its maquette is completely populated, Consort’s segment-maker performs meter rewriting. This proceeds in a more elaborate manner than the meter rewriting process as outlined in section 3.7 and subsection 3.9.4, and
involves a number of notable differences.

For reasons of computational efficiency, Consort rewrites the meters in each phrase of music annotating each performed timespan in the maquette before those phrases have even been inserted into the segment-maker’s score. Meter rewriting involves potentially many alterations to the contents of containers due to fusing and splitting, as well as many duration and offset lookups. Anytime a component is replaced or its duration changed, the cached offsets of components located in the score tree after the changed component as well as the durations of its parents are invalidated. They must be recomputed and re-cached on the next offset lookup performed on any component the score tree. Delaying inserting each inscribed timespan’s music into the segment-maker’s score guarantees that that music’s score depth remains shallow, and therefore limits the complexity of offset calculation during rewriting.

When beginning the meter rewriting process, the segment-maker converts its fitted meters into a timespan collection – Consort’s optimized timespan inventory class – containing timespans annotated with those fitted meters, one per timespan.

```python
>>> meters = metertools.MeterInventory([(3, 4), (2, 4), (6, 8), (5, 16)])
>>> meter_timespans = consort.SegmentMaker.meters_to_timespans(meters)
>>> print(format(meter_timespans))
consort.tools.TimespanCollection(
    [timespantools.AnnotatedTimespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(3, 4),
        annotation=metertools.Meter(
            '(3/4 (1/4 1/4 1/4))')
        ),
    ),
    timespantools.AnnotatedTimespan(
        start_offset=durationtools.Offset(3, 4),
        stop_offset=durationtools.Offset(5, 4),
        annotation=metertools.Meter(
            '(2/4 (1/4 1/4)')
        ),
    ),
    timespantools.AnnotatedTimespan(
        start_offset=durationtools.Offset(5, 4),
        stop_offset=durationtools.Offset(2, 1),
        annotation=metertools.Meter(
            '(6/8 ((3/8 (1/8 1/8 1/8)) (3/8 (1/8 1/8 1/8))))')
        ),
    ),
    timespantools.AnnotatedTimespan(
        start_offset=durationtools.Offset(2, 1),
        stop_offset=durationtools.Offset(37, 16),
        annotation=metertools.Meter(
            '(5/16 ((3/16 (1/16 1/16 1/16)) (2/16 (1/16 1/16)))')
```
Representing meters as timespans provides two important benefits. First, meters intersecting a given division within a phrase can be efficiently located using the search methods implemented on TimespanCollection. Second, the time relation methods implemented on Timespan can be used to test if a given division's timespan is congruent – that is, possesses an identical start and stop offset – to a meter's timespan. Divisions containing solely rests which are also congruent to a meter timespan can be rewritten with full-bar rests.

The segment-maker then proceeds through each demultiplexed timespan inventory in the maquette, iterating over each timespan, then over each division in that performed timespan’s music. Timespans whose rhythm-maker forbids meter rewriting – via the forbid_meter_rewriting flag on an optional DurationSpellingSpecifier – are skipped over.\(^3\) In order to determine which meter governs a division, that division’s timespan must be retrieved and then translated before it can be used to search the inventory of meter timespans. Because each phrase of music annotating each performed timespan has not yet been inserted into the score, they all express their start offset as 0. Likewise, each phrase’s child divisions express their start offsets relative to their parent phrase’s 0 start offset. Translating each division’s timespan relative to the start offset of the performed timespan annotated by that division’s phrase provides a useful search target for the meter timespan inventory. The translated division timespan represents the timespan that division would occupy if its phrase, and all other phrases, had already been inserted into the appropriate voice in the score. Intersecting meters can then be found through a simple search and retranslated relative to the current performed timespan’s start offset, giving their appropriate location within the not-yet-inserted phrase. The following operations outline the translation and search process:

```
>>> inscribed_timespan = consort.PerformedTimespan(...
    ...    start_offset=(5, 4),
    ...    stop_offset=(9, 5),
    ...    music=Container("{ c'4 }{ c'2 }{ c'4 }")
    ...
)
>>> division = inscribed_timespan.music[1]
>>> division_timespan = inspect_(division).get_timespan()
>>> print(format(division_timespan))

timespantools.Timespan(
    start_offset=durationtools.Offset(1, 4),
)
```

\(^3\)It may be undesirable to rewrite a rhythm’s meter in certain situations, particularly if a composer is trying to avoid the introduction of ties or dots for whatever reason.
With the appropriate meters selected, rewriting continues very much as described in subsection 3.9.4. Tuplets are rewritten solely with respect for the pre-prolated contents durations, and unprolated containers are rewritten with respect for their intersecting meter, with an initial offsets applied to the meter rewriting process if they happen to start later than their meter. Additionally, Consort’s meter rewriting tests silent meters – those whose leaves consist entirely of rests – for congruency with the current meter. Any division consisting solely of rests which also begins and ends at the start and stop offsets of a meter’s timespan can be rewritten instead as a single full-bar rest. The segment-maker also attaches a StaffLinesSpanner to the silent division, which collapses the staff down to a single line, giving the score a fragmented appearance. Finally, Consort’s segment-maker performs a logical-tie cleanup pass, fusing all 2-length logical ties consisting of matched pairs of $\frac{1}{16}$ or $\frac{1}{32}$ notes. This takes care of some possible artifacts of heavily subdivided meter-rewriting, and makes the final rhythmic output generally more readable.\footnote{Special thanks to Rei Nakamura for the suggestion.}
4.3.10 Populating the score

After meter-rewriting, the segment-maker can finally populate its score. To do so, it iterates through its timespan maquette and still-unpopulated score in parallel, extracting the inscribed music from each performed timespan in the maquette and inserting those phrases into the score in the appropriate voice. With the segment-maker’s score populated, rhythmic interpretation ends and non-rhythmic interpretation may begin.

4.4 Non-rhythmic interpretation

Non-rhythmic interpretation involves the process of progressively embellishing the contents of the rhythmically-interpreted score while preserving the score hierarchy and attack-point structure. During non-rhythmic interpretation, the segment-maker may attach grace notes, attach indicators and spanners, change the pitches of notes or even replace those notes with chords.

4.4.1 Score traversal

Recall the earlier discussion in subsection 2.6.2 regarding score iteration techniques. Of the various possible types of iteration, Consort’s handlers primarily make use of two during non-rhythmic interpretation. The first – attack-point or time-wise logical tie iteration – iterates through all logical ties in the score in time order according to their start offset in the score, regardless of their vertical position within the score. Logical ties with identical start offsets – those appearing at simultaneities across voices – are then sorted by their score order. This results in an iteration which moves first forward in time and then from the top of the score to the bottom. The second technique – phrase-wise iteration – locates each voice context in the score, then iterates through the top-level containers in that voice. These top-level containers are the same containers reference by each performed timespan’s music property, and represent each phrase in the maquette. Attack-point iteration helps maintain the continuity of some process across multiple voices in time, while phrase-wise iteration returns entire phrases and therefore allows processes to treat those phrases and all of the components located in the subtree rooted at that phrase as a single group.

Consider the following two voices score, produced by a segment-maker and annotated to show each division and phrase:

```plaintext
>>> musicSpecifier = consort.MusicSpecifier(  ...
    ... attachmentHandler=consort.AttachmentHandler(),
    ... rhythmMaker=rhythmmakertools.TaleaRhythmMaker(  ...
    ...    extraCountsPerDivision=(0, 1),
    ...    talea=rhythmmakertools.Talea([2, 3, 2, 4], 16),
```
At the beginning of non-rhythmic interpretation, the segment-maker constructs an **attack-point map**: a mapping of pitched logical ties against **attack-point signatures**. Consort’s `AttackPointSignature` class caches information about the location of each pitched logical tie in the segment, including its normalized position within its division, phrase and segment and its indices within the same. The segment-maker constructs the attack-point map by iterating over all note and chords in its score in time-wise order. It selects the logical tie for each iterated leaf, skipping those for which the leaf is not the logical tie’s head, and constructs an attack-point signature for the logical tie, storing both in an ordered dictionary. The segment-maker caches its attack-point map on its instance, allowing it to be referenced by its own methods, as well as examined after interpretation completes.
Because the attack-point map was constructed in time-wise order, each cached logical tie can be visited in time order when iterating over it. The following example iterates over the attack-point map, retrieving each key/value pair – comprised of a logical-tie selection and an attack-point signature – while enumerating them – producing the index of that pair, e.g. first, second, third, etc. – and attaches some markup to the head of that logical tie comprised of its index formatted within a padded box:

```
>>> for index, key_value_pair in enumerate(attack_point_map.items()):
...    logical_tie, attack_point_signature = key_value_pair
...    markup = Markup(index, Up)
...    markup = markup.smaller().pad_around(0.25).box()
...    attach(markup, logical_tie.head)
...
```

Attack-point signatures cache each logical-tie’s logical tie index and division index. A logical tie’s logical tie index gives its index within the list of all logical ties starting within its division. A logical tie’s division index give the index of the division within which it starts within the phrase itself. The following iteration shows the division index and
logical tie index of each attack-point, boxed and separated by a colon. The annotation brackets clarify the divisions within each phrase:

```python
>>> for logical_tie, attack_point_signature in attack_point_map.items():
...     for markup in inspect_(logical_tie.head).get_markup():
...         detached = detach(markup, logical_tie.head)
...         string = '(:):'.format(
...             attack_point_signature.division_index,
...             attack_point_signature.logical_tie_index,
...         )
...         markup = Markup(string, Up)
...         markup = markup.smaller().pad_around(0.25).box()
...         attach(markup, logical_tie.head)
...
>>> show(illustration)
```

Attack-point signatures maintain the position of each logical tie’s start offset normalized between 0 and 1 within its division, as well as its phrase. The following iteration show each logical tie’s phrase position as a fraction, after detaching the previously attached markup:

```python
>>> for logical_tie, attack_point_signature in attack_point_map.items():
...     for markup in inspect_(logical_tie.head).get_markup():
...         detached = detach(markup, logical_tie.head)
...         phrase_position = attack_point_signature.phrase_position
...         markup = Markup.fraction(phrase_position)
...         markup = Markup(markup, Up)
...         markup = markup.smaller().pad_around(0.25).box()
...         attach(markup, logical_tie.head)
...
>>> show(illustration)
```

197
Attack-point signatures also maintain the position of each logical tie’s start offset in the context of the entire segment’s timespan:

```python
>>> for logical_tie, attack_point_signature in attack_point_map.items():
...     for markup in inspect_(logical_tie.head).get_markup():
...         detached = detach(markup, logical_tie.head)
...         segment_position = attack_point_signature.segment_position
...         markup = Markup.fraction(segment_position)
...         markup = Markup(markup, Up)
...         markup = markup.smaller().pad_around(0.25).box()
...         attach(markup, logical_tie.head)
...     >>> show(illustration)
```

Consort’s phrase-wise iteration can be demonstrated by building an iterator function. The following function iterates through its score argument by voice, then iterates over the top-level containers within that voice. For each iterated container, it checks if that container’s effective music specifier is equivalent to the “silent” music specifier which Consort attaches to its silent phrases. If the music specifiers match, the phrase must be silent, and the function continues past it. If the music specifiers don’t match, the phrase must have some significant musical content, and therefore the function yields that phrase.
>>> def iterate_phrasewise(score):
    ...     prototype = consort.MusicSpecifier()
    ...     silent_music_specifier = consort.MusicSpecifier()
    ...     for voice in iterate(score).by_class(Voice):
    ...         for phrase in voice:
    ...             music_specifier = inspect_(phrase).get_effective(prototype)
    ...             if music_specifier == silent_music_specifier:
    ...                 continue
    ...             yield phrase

Before running the iterator to attach the phrase-index markup, the previously attached markup must be removed:

>>> for logical_tie in attack_point_map.keys():
    ...     for markup in inspect_(logical_tie.head).get_markup():
    ...         detached = detach(markup, logical_tie.head)

By chaining a call to the above-created `iterate_phrasewise()` function with Python’s built-in `enumerate()`, phrases can be extracted from the score along with their index. The result simply counts each non-silent phrase left-to-right in each voice, from the top of the score to the bottom:

>>> for index, phrase in enumerate(iterate_phrasewise(segment_maker.score)):
    ...     first_leaf = phrase.select_leaves()[0]
    ...     markup = Markup(index, Up)
    ...     markup = markup.smaller().pad_around(0.25).box()
    ...     attach(markup, first_leaf)
    ...

Some of the handlers used during non-rhythmic interpretation rely on information about each pitched logical tie in the score. Both grace- and pitch-handlers iterate through the score in time order. While grace-handlers do not make use of information stored within each logical tie’s associated attack-point signature – in the current implementation of Consort only pitch-handlers rely on attack-point signatures –, simply caching the time-order of all logical ties within the segment saves multiple complex iteration procedures.
4.4.2 Grace-handlers

Consort’s segment-maker can be instructed to attach grace notes to various parts of the score by defining a grace-handler on a music-specifier. Grace-handlers attach grace containers to logical ties within a score in a patterned way, according to a cyclic sequence of counts, as well as collection of flags which restrict what leaves may be selected for attachment. Abjad implements grace notes as normal leaves – notes, chords and rests – within special components which act both as Abjad Container classes as well as attachable indicators. That is to say, grace notes must be placed within one of these grace containers which is then attached to another leaf in the score much like any other indicator such as a dynamic indication or articulation. Grace-handlers traverse the score by logical tie in time-order, iterating over the previously cached ordered dictionary of attack-points which was generated at the beginning of non-rhythmic interpretation. Consider the following music specifier, whose grace-handler places grace notes with a cyclic count of 1, 2, 0, 0, 0.

```python
>>> music_specifier = consort.MusicSpecifier(
...     ...     grace_handler=consort.GraceHandler(
...     ...         ...         counts=(1, 2, 0, 0, 0),
...     ...     ),
...     ...     rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
...     ...         ...         extra_counts_per_division=(0, 1),
...     ...     ),
...     ...
... )
>>> timespan_maker = consort.TaleaTimespanMaker(
...     ...     initial_silence_talea=rhythmmakertools.Talea([0, 1], 4),
...     ...     playing_groupings=(1, 2, 2),
...     ...     playing_talea=rhythmmakertools.Talea([2, 3], 8),
...     ...     silence_talea=rhythmmakertools.Talea([1, 2, 3, 4], 8),
...     ...
... )
>>> music_setting = consort.MusicSetting(
...     ...     timespan_maker=timespan_maker,
...     ...     v1=music_specifier,
...     ...     v2=music_specifier,
...     ...
... )
>>> segment_maker = consort.SegmentMaker(
...     ...     desired_duration_in_seconds=8,
...     ...     discard_final_silence=True,
...     ...     permitted_time_signatures=[(2, 4), (5, 16), (3, 4)],
...     ...     score_template=templatetools.GroupedRhythmicStavesScoreTemplate(
...     ...         ...         staff_count=2,
...     ...         ...         with_clefs=True,
...     ...     ),
...     ...     settings=[music_setting],
...     ...     tempo=indicatortools.Tempo((1, 4), 72),
...     ...
... )
>>> show(segment_maker, verbose=False)
```
Note above that the first note in the score by time-wise ordering – the initial \( \frac{1}{16} \) note in the upper staff – does not receive the expected single grace note. The grace note pattern starts instead on the \( \frac{1}{8} \) note that follows. For purely practical reasons, Consort’s grace-handler will not put grace notes before the first leaf in any voice in a score. LilyPond’s grace spacing algorithm does not operate well with its strict proportional spacing algorithm. When spaced strictly proportionally, grace notes which are positioned before the beat – via LilyPond’s \texttt{\textbackslash grace}, \texttt{\textbackslash appoggiatura} or \texttt{\textbackslash acciaccatura} commands – may end up colliding with other glyphs in the staff. Consort works around this by using LilyPond’s \texttt{\textbackslash afterGrace} command, which places grace notes after the note to which they attach. This positioning allows grace notes to avoid most collisions. Unfortunately, this also means that grace notes must always appear after a note or rest, making it impossible to start a segment with grace notes.

4.4.3 Pitch-handlers

Consort’s pitch-handlers apply pitches to logical ties within a score in a patterned way, through the use of various cyclic sequences applied to each logical tie in time-wise order, much like grace-handlers. Consider the following short segment, containing two voices of \( \frac{1}{16} \) note rhythms interspersed with rests. Without any pitch-handler specified, both voices default to creating notes pitched at middle-C:

```python
>>> segment_maker = consort.SegmentMaker(
...     desired_duration_in_seconds=9,
...     #omit_stylesheets=True,#
...     permitted_time_signatures=[(3, 4)],
...     score_template=template.GroupedStavesScoreTemplate(
...         staff_count=2,
...     ),
...     tempo=indicator.Tempo((1, 4), 60),
... )
>>> music_specifier = consort.MusicSpecifier(
...     rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
...         talea=rhythmmakertools.Talea([1], 16),
...     ),
... )
>>> timespan_maker = consort.TaleaTimespanMaker(
```
By reconfiguring the above music specifier with an \texttt{AbsolutePitchHandler} – a concrete subclass of Consort’s abstract \texttt{PitchHandler} – different pitches can be applied to each note in the segment. The following pitch-handler paints a C-major scale across all of the logical ties in the resulting segment. Note how the G\textsubscript{4}, A\textsubscript{4}, B\textsubscript{4} and C\textsubscript{5} of the applied C major scale alternate between the voices according to both their time-wise position and their score order, as discussed in subsection 4.4.1:

$$\text{\begin{verbatim}
>>> music_specifier = new(
...     music_specifier,
...     pitch_handler=consort.AbsolutePitchHandler(
...         pitchSpecifier="c' d' e' f' g' a' b' c'\'',
...     ),
... )
>>> music_setting = consort.MusicSetting(
...     timespan_maker=timespan_maker,
...     v1=music_specifier,
...     v2=music_specifier,
... )
>>> segment_maker = new(segment_maker, settings=[music_setting])
>>> show(segment_maker, verbose=False)
\end{verbatim}}$$

$$\text{\begin{verbatim}\frac{3}{4} \text{= 60}
\end{verbatim}}$$
Pitch-handlers apply their pitch patterns in time-wise order regardless of rhythmic texture:

```
>>> music_specifier = new(
...     music_specifier,
...     rhythm_maker__talea__counts=[1, 2, 3, 4],
... )
>>> music_setting = consort.MusicSetting(
...     timespan_maker=timespan_maker,
...     v1=music_specifier,
...     v2=music_specifier,
... )
>>> segment_maker = new(segment_maker, settings=[music_setting])
>>> show(segment_maker, verbose=False)
```

As discussed in subsection 4.3.5, pitch-handlers, like rhythm-makers, maintain a seed value keyed to the music specifier in which they are defined, allowing them to maintain their pattern even in fragmentary or interrupted musical textures. Note how, when interrupted by phrases defined by a different music specifier, the previously defined texture continues its pitch application pattern after each interruption or silence exactly where it left off. For example, the A₄ 1/16 note in the lower voice in measure one is continued by the B₄ 1/8 note in the upper voice at the beginning of measure two, and followed by the C₅ 1/8 note directly below in the lower voice. The C-major pattern continues despite the intrusion of the triplet texture defined by the other_music_specifier:

```
>>> other_music_specifier = consort.MusicSpecifier(
...     pitch_handler=consort.AbsolutePitchHandler(pitch_specifier='g fs e f'),
...     rhythm_maker=rhythmmakertools.EvenDivisionRhythmMaker(
...         denominators=[8],
...         extra_counts_per_division=(1,),
...     ),
... )
>>> other_music_setting = consort.MusicSetting(
...     timespan_maker=consort.TaleaTimespanMaker(
...         initial_silence_talea=rhythmmakertools.Talea([1], 2),
...         silence_talea=rhythmmakertools.Talea([1], 2),
...     ),
...     v1=other_music_specifier,
...     v2=other_music_specifier,
... )
>>> segment_maker = new(
```

203
Pitch-handlers may define explicit *pitch specifiers*, which behave somewhat analogously to the ratio-parts expressions discussed in subsection 4.3.1, describing which pitches or pitch-classes are to be used in which sections of a segment, as partitioned by a ratio:

```python
>>> pitch_specifier = consort.PitchSpecifier(
...     pitch_segments=(
...         "c' e' g'",
...         "fs' g' a'!",
...         "b d'!",
...     ),
...     ratio=(1, 2, 3),
... )
>>> pitch_choice_timespans = consortPitchHandler.get_pitch_choice_timespans(
...     pitch_specifier=pitch_specifier,
...     duration=segment_maker.desired_duration,
... )
>>> print(format(pitch_choice_timespans))
consort.tools.TimespanCollection(
[  
timespantools.AnnotatedTimespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(3, 8),
    annotation=datastructuretools.CyclicTuple(
        [pitchtools.NamedPitch("c'")],
        [pitchtools.NamedPitch("e'")],
        [pitchtools.NamedPitch("g'")],
    ),
),
  timespantools.AnnotatedTimespan(
    start_offset=durationtools.Offset(3, 8),
    stop_offset=durationtools.Offset(9, 8),
    annotation=datastructuretools.CyclicTuple(
        [pitchtools.NamedPitch("fs'")],
        [pitchtools.NamedPitch("g'")],
        [pitchtools.NamedPitch("a'!")],
    ),
),
]"
When used to reconfigure the previously-defined music specifier's pitch-handler, each of the pitch specifier's pitch segments appears in only one portion of the resulting score, as defined by the pitch-choice timespan inventory:

```python
>>> music_specifier = new(
    ...     music_specifier,
    ...     pitch_handler__pitch_specifier=pitch_specifier,
    ... )
>>> music_setting = consort.MusicSetting(
    ...     timespan_maker=timespan_maker,
    ...     v1=music_specifier,
    ...     v2=music_specifier,
    ... )
>>> segment_maker = new(segment_maker, settings=[music_setting])
>>> show(segment_maker, verbose=False)
```

Likewise, a pitch-handler may define a pitch-operation specifier, which behaves similarly to pitch specifiers. Pitch-operation specifiers combine a sequence of pitch-operations – transposition, inversion, retrogression, rotation, etc. – with a ratio defining where those operations should appear during the course of a segment. Pitch-operation specifiers may be used alongside pitch specifiers, and with differing ratios. The following pitch-operation specifier
partitions the segment into three unequal parts, applying an operation to the first \(\frac{1}{4}\), no operation to the middle \(\frac{1}{2}\), and another operation to the final \(\frac{1}{4}\) of the segment:

```python
>>> music_specifier = new(
    ...     pitch_handler_pitch_operation_specifier=consort.PitchOperationSpecifier(
    ...         pitch_operations=(
    ...             pitchtools.Inversion(),
    ...         ),
    ...     ),
    ...     ratio=(1, 2, 1),
    ... ),
>>> music_setting = consort.MusicSetting(
    ...     timespan_maker=timespan_maker,
    ...     v1=music_specifier,
    ...     v2=music_specifier,
    ... ),
>>> segment_maker = new(segment_maker, settings=[music_setting])
>>> show(segment_maker, verbose=False)
```

Pitch-handlers may also be configured to forbid immediate pitch repetitions within the same voice via the `forbid_repetitions` flag, assuming the current logical tie has more than one pitch available via its pitch-choices:

```python
>>> music_specifier = new(
    ...     pitch_handler_forbid_repetitions=True,
    ... ),
>>> music_setting = consort.MusicSetting(
    ...     timespan_maker=timespan_maker,
    ...     v1=music_specifier,
    ...     v2=music_specifier,
    ... ),
>>> segment_maker = new(segment_maker, settings=[music_setting])
>>> show(segment_maker, verbose=False)
```
Chords may be applied in a patterned way to the logical ties a pitch-handler iterates over by defining a sequence of *logical tie expressions* – classes which, when called on a logical tie, modify or replace each of the leaves in that logical tie, leaving the tie itself in place. Consort provides a few subclasses of logical tie expression, such as the ChordExpression, HarmonicExpression and KeyClusterExpression. When called on a logical tie, chord expression replace each note in the tie with a chord whose pitches may be specified either absolutely or in terms of intervals relative to the original pitch. Note that Python’s `None`, like in the previously defined pitch-operation specifier, acts as a “no-op”, making no change at all:

```python
>>> music_specifier = new(
...     music_specifier,
...     pitch_handler__logical_tie_expressions=
...     (consort.ChordExpression(chord_expr=(-2, 0, 2)),
...      consort.ChordExpression(chord_expr=(-7, 0, 7)),
...      None,
...     ),
... )

>>> music_setting = consort.MusicSetting(
...     timespan_maker=timespan_maker,
...     v1=music_specifier,
...     v2=music_specifier,
... )

>>> segment_maker = new(segment_maker, settings=[music_setting])

>>> show(segment_maker, verbose=False)
```

All of the above pattern definitions – pitch specifiers, pitch-operation specifiers and sequence of logical tie expressions – continue to apply in order even over an interrupted or otherwise fragmentary texture:
Now, consider again the $\frac{3}{16}$ note music specifier from earlier in this section, whose pitch-handler cycled through a C-major scale. Pitch-handlers may be configure to use the same pitch for each division, or even phrase, rather than choosing a new pitch for each encountered logical tie. This behavior is defined by the pitch-handler's *application rate*, which is quite analogous to the music specifier sequence application rate discussed in subsection 4.1.3. The following pitch handler applies the same pitch from its pitch specifier to each logical tie within each division. Both the divisions and phrases have been annotated for clarity:

```python
>>> music_specifier = consort.MusicSpecifier(
    ...    pitch_handler=consort.AbsolutePitchHandler(
    ...       pitch_application_rate='division',
    ...       pitch_specifier='c' 'd' 'e' 'f' 'g' 'a' 'b' 'c''',
    ...    ),
    ...    rhythm_maker=rhythmmakertools.EvenDivisionRhythmMaker(denominators=[16]),
    ...)
>>> music_setting = consort.MusicSetting(
    ...    timespan_maker=timespan_maker,
    ...    v1=music_specifier,
    ...    v2=music_specifier,
    ...)
>>> segment_maker = new(segment_maker, settings=[music_setting])
>>> lilypond_file, metadata = segment_maker(verbos=False)
>>> consort.annotate(lilypond_file.score)
>>> show(lilypond_file)
```
Per-division and per-phrase pitch application rates rely on the attack-point signature objects collected at the beginning of non-rhythmic interpretation to determine if any encountered logical tie is the first of its division or phrase. Pitch-handlers also require more complex seed-tracking logic – maintained by a dedicated `SeedSession` class –, as each encountered logical tie may not require choosing a new pitch, but could still require choosing a new logical tie expression. The above music specifier can be re-configured to apply the same pitch to every logical tie in a phrase:

```python
>>> music_specifier = new(
    ...     music_specifier,
    ...     pitch_handler__pitch_application_rate='phrase',
    ... )
>>> music_setting = consort.MusicSetting(
    ...     timespan_maker=timespan_maker,
    ...     v1=music_specifier,
    ...     v2=music_specifier,
    ... )
>>> segment_maker = new(segment_maker, settings=[music_setting])
>>> lilypond_file, metadata = segment_maker(annotate=True, verbose=False)
>>> show(lilypond_file)
```

Additionally, a pattern of interval deviations may be applied on top of any chosen pitches, regardless of the pitch-handler’s application rate. This provides a mechanism for intermittently varying an otherwise stable melodic contour:
Other pitch-handlers are possible. For example, Consort’s PitchClassPitchHandler first applies pitch-classes to each logical tie, then an octavation according to a register specifier which acts similarly to a break-point function, shifting the pitch-class up and down by octaves according to the shape of the specifier and the logical tie’s position within its division, phrase and segment. Pitch-handlers based on vertical sonorities, spectral information, or various other techniques could also be implemented. As it stands, I have so far only implemented two for the pieces I have written with Consort, finding them sufficient for now.

4.4.4 Attachment-handlers

Attachment-handlers manage the process of attaching indicators and spanners to selections of components within a score. Unlike grace-handlers and pitch-handlers, they iterate over the score by voice and phrase, rather than by logical-tie time-wise. Phrase-wise iteration allows attachment-handlers to create highly-contextualized attachments, considering each component in a phrase in terms of each other component. Attachment-handlers aggregate attachment expressions, objects pairing a component selector – as described in subsection 2.6.3 – and an iterable of attachments – indicators and spanners – or component expressions. Selectors chain together a series of callbacks which progressively refine a selection as each callback processes it. Much like the other handler classes described already, an attachment-expression’s attachments sequence cycles, allowing different groups of attachments to be attached
each time the attachment-expression is called with a different seed value. Attachment-handlers associate their attachment expressions with underscore-delimited string keys. This mechanism, nearly identical to that employed by music settings for associating music specifiers with voice-name abbreviations, allows attachment expressions – which may have an arbitrary number of such associations – to be reconfigured through templating to add new attachment expressions or overwrite or nullify specific existing expressions.\(^5\)

Consider the following two-staff segment containing phrases interspersed by rests and a surface-rhythm with a variety of tuplets and ties:

```python
>>> music_specifier = consort.MusicSpecifier(
...     attachment_handler=consort.AttachmentHandler(),
...     rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
...         extra_counts_per_division=(0, 1),
...         talea=rhythmmakertools.Talea([1, 2, 3, 1, 4], 16),
...     ),
... )
>>> timespan_maker = consort.TaleaTimespanMaker(
...     initial_silence_talea=rhythmmakertools.Talea([0, 1], 4),
...     playing_groupings=(1, 2, 2),
...     playing_talea=rhythmmakertools.Talea([2, 3], 8),
...     silence_talea=rhythmmakertools.Talea([1, 2, 3, 4], 8),
... )
>>> music_setting = consort.MusicSetting(
...     timespan_maker=timespan_maker,
...     v1=music_specifier,
...     v2=music_specifier,
... )
>>> segment_maker = consort.SegmentMaker(
...     desired_duration_in_seconds=8,
...     discard_final_silence=True,
...     permitted_time_signatures=[[2, 4], [5, 16], [3, 4]],
...     score_template=templatetools.GroupedRhythmicStavesScoreTemplate(
...         staff_count=2,
...         with_clefs=True,
...     ),
...     settings=[music_setting],
...     tempo=indicatortools.Tempo((1, 4), 72),
... )
>>> show(segment_maker, verbose=False)
```

\(^5\)The principle employed by both music settings and attachment handlers is crucial: named references beat positional references for ease of reconfiguration.
Accents can be attached to the first leaf of each phrase by configuring the music specifier’s attachment-handler with an attachment expression whose attachments are simply an instance of an Abjad Articulation and whose selector has been configured to first select all leaves and then select the first item in that selection of leaves: the first leaf:

```python
>>> attachment_expression = consort.AttachmentExpression(
    ...     attachments=Articulation('accent'),
    ...     selector=selectortools.Selector().by_leaves()[0],
    ... )
>>> musicSpecifier = new(
    ...     musicSpecifier,
    ...     attachment_handler__accents=attachment_expression,
    ... )
>>> music_setting = new(
    ...     music_setting,
    ...     v1=musicSpecifier,
    ...     v2=musicSpecifier,
    ... )
>>> segment_maker = new(segment_maker, settings=[music_setting])
>>> show(segment_maker, verbose=False)
```

Tenuto articulations can be attached to the head of every logical tie in each phrase – except the very first – with a more complex selector. The following tenuto-attaching attachment-expression’s selector first selects all leaves in each phrase, then selects all pitched logical ties, then selects all but the first of those logical ties – rather than all but the first leaf in each of those logical ties – and finally selects the first leaf in each selected logical tie:

```python
>>> musicSpecifier = new(
    ...     musicSpecifier,
    ...     attachment_handler__tenuti=consort.AttachmentExpression(
    ...         attachments=Articulation('tenuto'),
    ...         selector=selectortools.Selector()
    ... )
```

212
Attaching a spanner to the entirety of a phrase is a much simpler process. When configuring an attachment-handler with a new key, if the value of that key is not already an attachment expression, the attachment-handler creates a new attachment expression and sets the value as the attachment expression’s attachment list. Likewise, an attachment expression which has no selector defined uses a default selector when called, which simply selects whatever the attachment expression was called against. In the following reconfiguration, a slur is attached to the entirety of each phrase, using the configuration coercion described above:

```python
>>> music_specifier = new(
...     music_specifier,
...     attachment_handler__slurs=Slur()
... )
>>> music_setting = new(
...     music_setting,
...     v1=music_specifier,
...     v2=music_specifier,
... )
>>> segment_maker = new(segment_maker, settings=[music_setting])
>>> show(segment_maker, verbose=False)
```
Attachment-handlers and their attachment expressions can also delegate to objects specifically designed for attaching indicators and spanners, much like the logical tie expressions outlined in subsection 4.4.3. For example, Consort's *dynamic expression* attaches dynamic indications, potentially spanned by hairpins, to the first leaf in each division of a phrase, and to the last leaf of the phrase as well, taking care of special cases such as divisions with insufficient numbers of leaves. Dynamic expressions encapsulate their own selection logic which is generally too difficult to implement simply through chaining component selector callbacks. Consider the locations of the dynamics in the following annotated segment:

```plaintext
>>> music_specifier = new(
  ... music_specifier,
  ... attachment_handler__dynamics=consort.DynamicExpression(['f', 'p'])
  ... )
>>> music_setting = new(
  ... music_setting,
  ... v1=music_specifier,
  ... v2=music_specifier,
  ... )
>>> segment_maker = new(segment_maker, settings=[music_setting])
>>> lilypond_file, metadata = segment_maker(annotate=True, verbose=False)
>>> show(lilypond_file)
```

Note above how each division in each phrase begins with a dynamic, and the last division in each phrase – including those which are the *only* division – contain two dynamics, with the exception of those that only contain a single
4.4.5 Expressive Attachments

Complex attachment-based formatting scenarios can also be constructed without relying on component expressions such as DynamicExpression. One technique used extensively in my scores, especially Invisible Cities (ii): Armilla, is to attach various idiomatic indicators as non-formatting annotations to components throughout each phrase, and then to attach a specially-designed spanner to the entirety of each phrase which knows how to inspect the leaves it covers for the previously attached annotations. For example, Consort’s StringContactSpanner can inspect its leaves for StringContactPoint indicators and then construct markup for each contact point – sul ponticello, ordinario, and so forth – bridged by arrows when the contact points change, and parenthesizing the contact point markup when cautionary. Likewise, the BowContactSpanner can inspect its leaves for BowContactPoint indicators – which indicate the current position along the hair of the bow – as well as BowMotionTechnique indicators – which describe various motion qualities like jeté and tremolo – replacing the note heads of the staff with fractions for the current bow position and adding up- or down-bow markup as necessary. When combined with the appropriate typographic overrides, complex graphic notation results:

```python
>>> staff = Staff()
>>> staff.extend(r"c'4. c'8 \times 2/3 { c'4 c'4 c'4 }")

>>> leaves = staff.select_leaves()
>>> attach(indicatortools.BowMotionTechnique('jete'), leaves[0])
>>> attach(indicatortools.BowContactPoint((1, 4)), leaves[0])
>>> attach(indicatortools.BowContactPoint((3, 4)), leaves[1])
>>> attach(indicatortools.BowContactPoint((1, 2)), leaves[2])
>>> attach(indicatortools.BowMotionTechnique('circular'), leaves[3])
>>> attach(indicatortools.BowContactPoint((1, 1)), leaves[3])
>>> attach(indicatortools.BowContactPoint((0, 1)), leaves[4])

>>> attach(Clef('percussion'), staff)
>>> override(staff).bar_line.transparent = True
>>> override(staff).dots.staff_position = -8
>>> override(staff).flag.Y_offset = -8.5
>>> override(staff).glissando.bound_details__left__padding = 1.5
>>> override(staff).glissando.bound_details__right__padding = 1.5
>>> override(staff).glissando.thickness = 2
>>> override(staff).script.staff_padding = 3
>>> override(staff).staff_symbol.transparent = True
>>> override(staff).stem.direction = Down
>>> override(staff).stem.length = 8
>>> override(staff).stem.stem_begin_position = -9
>>> override(staff).time_signature.stencil = False
```
Notation created in this way treats spanners as typographic post-processors. The positioning of the various bowing indicators – the bow motion technique indicators, the bow contact point indicators – constitutes the compositional act, and the bookkeeping inherent to drawing the correctly formatted lines between them simply constitutes one part of the automated typesetting process.

4.4.6 Post-processing

With grace-, pitch- and attachment-handling completed, non-rhythmic interpretation proceeds to its final step – post-processing –, during which various typographic adjustments are made to the score.

First, Consort’s segment-maker creates a floating time-signature context – described in subsection 3.9.5 – and inserts it as the first child of the score, assuming the score template did not already provide for one. The segment-maker then populates its time-signature context with measures containing typographic spacer-skips, creating the appearance of time signature indications floating above each bar-line at the top of the score. The segment-maker attaches other objects to this context as well, such as the its tempo indication and any configured rehearsal mark or segment name, e.g. “Interlude...”. Segments can also be configured with repeat signs or final bar-line indications. Consort’s segment-maker attaches such indicators to the appropriate leaves in the score during post-processing. Likewise, if a segment has been configured as the final segment, any colophon information – the locations and dates of composition – will be added beneath the last measure of the score.

More complex post-processing may also be carried out. For reasons related solely to how LilyPond handles various typographic constructs during typesetting, it may prove necessary to copy certain voices in the score, maintaining all rhythmic information therein, but changing their context name and filtering out various spanners and indicators so that only a specific subset of typographic commands remain. In doing so, it is possible to isolate typographic effects to each voice performed “simultaneously” in a staff. For example, my score Invisible Cities (ii): Armilla employs this voice-copying technique to create its bowing-staff typography. Up until the post-processing step of non-rhythmic interpretation, Armilla’s bowing voices contain a rhythm with many indicators and spanners attached – bow contact points, string contact points, bow motion techniques, bow contact spanners, string contact...
spanners, and so forth. Bow contact spanners and string contact spanners create conflicting typographic overrides when formatted, so it is necessary to duplicate the bowing voice multiple times, maintaining only the relevant indicators and spanners in each duplicate. By subclassing Consort’s built-in segment-maker Armilla can extend its score post-processing with an additional step consisting of these voice copying operations.

Last, the segment-maker wraps the score inside a LilyPondFile instance – Abjad’s object model for LilyPond input files – and configures the LilyPond file with any necessary stylesheet file includes. By subclassing Consort’s segment-maker, individual score packages can teach their segment-maker subclasses to automatically locate their stylesheet files relative to the package where that segment-maker subclass was defined. Such stylesheets should include context definitions for the TimeSignatureContext as well as any other custom contexts defined for the score.

As a final test, the segment-maker runs Abjad’s battery of component well-formedness checks against the score. These checks look for score-structure errors such as components appearing in multiple places in the score, ties which join notes with differing pitches, overlapping glissandi, or components with null parent references. If all tests pass, the segment-maker returns the LilyPond file object. Interpretation is complete.

4.5 Persistence & visualization

Once interpreted, a segment-maker’s illustration may be persisted to disk as LilyPond syntax for inclusion in other LilyPond files, rendered as a PDF for viewing, or even serialized for other purposes. Composers study the results of interpretation, make changes to each segment’s specification, and re-interpret as necessary, a large-scale re-enactment of interactive programming’s pervasive read-eval-print loop paradigm.
This page intentionally left blank.
Throughout the previous chapters, I have presented concepts, techniques, classes and functions for modeling notation and composition in code, accompanied by numerous visualizations, all executed in the context of an interactive Python interpreter session. In light of that, it’s important to understand that when composing computationally, I rarely work with the interactive console. Interactive sessions demonstrate simple techniques well, and allow for inspection of live code objects, but are prone to typing errors – consider the difficulty of typing dozens of lines perfectly – and ultimately discourage an iterative workflow.

Iteration is crucial for composing large-scale works, and requires that code and other assets be preserved throughout each cycle of definition and visualization so that they can be progressively revised, amended and otherwise reconfigured. Code must be saved to disk as modules in order to be executed or imported into other code. Likewise, code should be encapsulated for both manageability and legibility, preferably organized into multiple files articu-
lating their structure or purpose. Of course, once code is arranged as files on disk, a whole variety of additional questions emerge: How should that code be organized? In how many files? How much and which code in each each file? Where should the files be kept? Does the code create other assets? Where are these stored? How are they named? How is the code executed? How is the project structure maintained? How are changes to the project saved, and how does one compare one version to another? And how is the code in the project tested to guarantee that it works at all, let alone creates the musical result one expects and desires?

The answers to such questions – and there often are clear answers – revolve the practicalities of managing not simply a musical endeavor, but a software project: code and directory structuring, development tools and workflows, version control and testing. All of these are discussed, in greater or lesser detail, in the following sections.

5.1 Score directory layout

When working computationally, a structured, standardized approach to arranging one’s compositional workspace clarifies both the way one thinks about the act of composing, as well as the more menial workflows of document preparation which are inseparable from score-based composition. Standardization also allows one to reuse tools designed to manage one score package on another. Each of the Invisible Cities scores included in part II of this dissertation are implemented as Python packages extending both Abjad’s model of notation and Consort’s model of composition. Likewise, each Invisible Cities score package is structured into a nearly identical arrangement of eight top-level directories, each with a clearly delineated purpose and substructure.

ersilia/
   _init_.py .............................................. The score package Python initializer.
   build/ ................................ LilyPond and LaTeX files for building document targets.
   distribution/ ................................... Finished PDFs for performers and conductors.
   etc/ ....................................................... Notes, to-do lists and plans.
   makers/ ................................... Customized segment-makers, score templates and other classes.
   materials/ .................................. Materials used to configure segment-makers.
   segments/ ................................ Configured segment-makers and their illustrations.
   stylesheets/ ....................................... LilyPond stylesheets.
   test/ ....................................................... The score package test suite.

Figure 5.1: Ersilia’s top-level directory layout.

The top-level directories named in figure 5.1 house specific collections of assets used during composition or document preparation. Note the presence of an _init_.py file. This signals to Python that this directory represents an importable Python package. Assuming Python is aware of the location of the score, it and any further subpackages
Any subdirectories of `ersilia` also containing `__init__.py` files can be imported into the namespace of `ersilia`, allowing for the structured organization of any code assets employed by the score. Consider the various kinds of objects discussed in earlier chapters. Consort very broadly groups the objects used during composition into `makers`, `materials` and `segments`. Makers comprise any classes used during composition, such as segment-maker, pitch-handler or music specifier classes, but not their instances. Materials comprise instances of classes representing `out-of-time` musical constructs, such as pitch segments or music specifiers which have not yet been deployed along the timeline of the score. Segments comprise configured instances of the score’s segment-maker class which maquette together previously-defined materials `in-time` into a musical chronology. Each of these three categories of objects has its own directory in the directory structure outlined here: `makers/`, `materials/` and `segments/`. Additionally, materials and segments may be illustrable. Any associated illustration assets for those code objects is necessarily grouped together with its originating Python modules.

The document preparation process also involves a number of different types of assets and tasks. Any LilyPond-typeset PDF created during the composition of a score, such as a material or segment illustration, or even the final score itself likely requires a corresponding score-specific stylesheet containing the typographic overrides and LilyPond context definitions pertinent to that score. These stylesheets reside in the top-level `stylesheets/` directory. Scores composed with Abjad and Consort and typeset with LilyPond and LaTeX involve potentially many source files: LilyPond sources for the musical content, yet more LilyPond sources for concatenating and styling that content, LaTeX sources for cover pages, prefaces and performance notes, and still more LaTeX sources for concatenating the PDFs created while typesetting various other sources. These source files, in their various stages of typesetting, occupy the `build/` directory. Finally, the finished documents, ready to be delivered to ensembles, occupy the `distribution/` directory.

5.1.1 Makers

Each score’s `makers/` directory houses classes specific to that score, with each class stored on disk in its own Python module of the same name. In the case of my three `Invisible Cities` scores, these makers always comprise subclasses of Consort’s `SegmentMaker` and `ScoreTemplate` classes. The segment-maker subclasses effectively pre-load common information about the score, such as what time signatures to permit by default or what the end-of-score markup
should look like. They may also define a considerable amount of additional score post-processing, as in the case of Armilla, where many passes of voice copying were required to create the typography for the bowing staves. A score’s score template class necessarily defines the score’s instrumentation and context hierarchy, giving the order and grouping of each performer’s staff. As none of the scores in this dissertation make use of common instrumentations, each requires that a completely new score template be defined.

Any other class definitions required by the score package should also be placed into the makers/ directory. For example, Plague Water, whose directory structure is effectively identical to the Invisible Cities series but whose composition model both predated and heavily informed the development of Consort, houses nearly thirty separate class definitions in its makers/ directory.

### 5.1.2 Material packages

Materials represent *out-of-time* musical objects, such as pitch collections, rhythm-makers, and even fully configured music specifiers, which may appear at some point in the time-line of a piece. They are implemented as importable Python packages, grouped flatly into the score package’s top-level materials/ directory, itself a Python package due to the presence of an initializer file. Like the makers modules, material definition module define only a single object.

```plaintext
materials/
  __init__.py .......................................................... The subpackage initializer.
  abbreviations/ .......................................................... A material package.
  dense_timespan_maker/ ................................................ Another material package.
  guitar_agitato_musicSpecifier/ ..................................... Yet another material package.
  guitar_strummed_musicSpecifier/
  guitar_tremolo_musicSpecifier/
  guitar_undulation_tremolo_musicSpecifier/
  percussion_bamboo_windchimes_musicSpecifier/
  percussion_crotale_flash_musicSpecifier/
  percussion_crotale_interruption_musicSpecifier/
  ...
  .......................................................... Many more material packages.
```

Each material package contains at least its own initializer as well as a definition.py file consisting of Python code which defines or configures that material. Crucially, the actual code object expressed in that definition.py should be named after the material package itself. This simplifies the Python import process a great deal. Simple importing utilities can then be written to iterate over every directory within the materials/ directory, producing corresponding Python import commands of the form `from my_material.definition import my_material`. Automatically importing the objects defined in each material package into the materials namespace allows those objects to
be easily referenced from within each segment definition:

```python
>>> materials_names = dir(ersilia.materials)
>>> materials_names = [x for x in materials_names if not x.startswith('_')]
>>> print(materials_names)
['abbreviations', 'dense_timespan_maker', 'guitar_agitato_music_specifier',
'guitar_continuo_music_specifier', 'guitar_pointillist_harmonics_music_specifier',
'guitar_strummed_music_specifier', 'guitar_tremolo_music_specifier',
'guitar_undulation_tremolo_music_specifier', 'percussion_bamboo_windchimes_music_specifier',
'percussion_crotales_flash_music_specifier', 'percussion_crotales_interruption_music_specifier',
'percussion_low_pedal_music_specifier', 'percussion_marimba_agitato_music_specifier',
'percussion_marimba_ostinato_music_specifier', 'percussion_marimba_tremolo_music_specifier',
'percussion_snare_interruption_music_specifier', 'percussion_temple_block_fanfare_music_specifier',
'percussion_tom_fanfare_music_specifier', 'permitted_time_signatures', 'piano_agitato_music_specifier',
'piano_arm_cluster_music_specifier', 'piano_glissando_music_specifier',
'piano_palm_cluster_music_specifier', 'piano_pedals_music_setting', 'piano_pointillist_music_specifier',
'piano_string_glissando_music_specifier', 'piano_tremolo_music_specifier', 'pitch_pipe_music_specifier',
'saxophone_agitato_music_specifier', 'shaker_decelerando_music_specifier',
'shaker_sporadic_music_specifier', 'shaker_tremolo_music_specifier', 'sparse_timespan_maker',
'string_agitato_music_specifier', 'string_legato_music_specifier', 'string_low_pedal_music_specifier',
'string_overpressure_music_specifier', 'string_pointillist_music_specifier', 'string_tremolo_music_specifier', 'sustained_timespan_maker',
'tutti_timespan_maker', 'wind_agitato_music_specifier', 'wind_continuo_music_specifier',
'wind_low_pedal_music_specifier', 'wind_ostinato_music_specifier', 'wind_pointillist_music_specifier',
'wind_tremolo_music_specifier']
```

Figure 5.3: Directory listing of Ersilia's guitar_tremolo_music_specifier material package.

Materials – depending on their type – may be illustrable, as described in section 2.1. Any generated illustration assets – comprising a LilyPond input file and its resulting PDF – are stored in the material package alongside their originating definition module. The means by which these illustration files come to reside there is elaborated on in subsection 5.2.2.

### 5.1.3 Segment packages

Segments, like materials, are implemented as Python packages, grouped together into the top-level segments/ directory. Each segment package can be imported, has an obligatory Python initializer, as well as a definition module containing the definition of that segment’s segment-maker. Because Consort’s segment-maker class is illustrable, each segment package eventually houses that segment’s illustration LilyPond source and output PDF.
Each segment package may also house a `__metadata__.py` module, which stores information about that segment such as the number of measures it contains, the final time signature and tempo, and so forth. This information is generated automatically during the segment-maker's interpretation, and allows other segment-makers in other segment packages to draw conclusions about their own segment's context in the full score without requiring the re-interpretation of any other segments. Like the illustration sources, these are discussed in more depth in subsection 5.2.2.

A `__metadata__.py` module sibling to each segment package simply defines the order of segments. For example, the order of segments in *Ersilia* is `komokome`, `cut_1`, `sort`, `cut_2`, then `chemish`. Such an order must be declared explicitly as it cannot be ascertained from the lexical ordering of the names of the segment packages. The `segments/`-local metadata module allows for such an explicit ordering.

### 5.1.4 The stylesheets/ directory

The `stylesheets/` directory consists of LilyPond files containing typographic overrides, context definitions, document header markup, page layout configuration and Scheme function definitions. The file `stylesheet.ily` represents the primary stylesheet for the entire score, and contains most of the typographic customization. However, LilyPond stylesheets can *cascade*. Multiple stylesheets can be included into the same master score file with definitions in subsequently included stylesheets overriding those in earlier ones. Likewise, stylesheets – because they are simply LilyPond files – can be included directly into one another.
For example, in *Ersilia*, the master stylesheet file `stylesheet.ily` directly includes Scheme definitions from the file `scheme.ily`. That master stylesheet is included into the interpreted LilyPond source of every segment. However, segments beyond the first also include the stylesheet file `nonfirst-segment.ily` which suppresses the appearance of titles and other headers at the top of those segment illustrations, giving the impression when viewing those segments' illustrations alone that one has jumped into the middle of the full score.

```
stylesheets/
  nonfirst-segment.ily ................ Style information for segments after the first segment.
  parts-landscape.ily .................. Style information for landscape-orientation parts.
  parts-portrait.ily ................... Style information for portrait-orientation parts.
  scheme.ily ......................... LilyPond Scheme commands to be included in the primary stylesheet.
  stylesheet.ily ........................ The primary stylesheet.
```

*Figure 5.5: Directory listing of *Ersilia*’s *stylesheets/* directory.*

Because LilyPond supports the inclusion of one LilyPond source file into another, it is possible to separate the musical content of a work from the settings defining its typographic presentation or page layout. This is the approach I have taken, out of necessity, when building scores with Abjad and Consort. It is certainly possible to define every one of the typographic overrides found in `stylesheet.ily` in Abjad via its top-level `override()` and `set_()` functions. However, typesetting the illustration source of a segment in LilyPond generally takes less time than for that segment to be interpreted in Python. By separating out typographic overwrites from the musical “content” it becomes easier to iterate over refining those overrides by simply adjusting the stylesheet by hand and recompiling the already-interpreted segment illustration sources.

### 5.1.5 The build/ directory

The `build/` directory, outlined in figure 5.6, holds files pertinent to building scores and parts, along within any component documents, including front and back covers, prefaces or performance notes. The contents of the `build/` directory are organized into *document targets* and *assets*. Each document target subdirectory consists of source files for producing scores and parts in a particular format, such as tabloid or A4 paper, or for a particular performance or language translation. Assets include any LilyPond and LaTeX file consisting of textual or musical content to be included into a document target. For example, segment illustration LilyPond sources are housed in `build/segments/` while LaTeX includes – containing blocks of prose to be flowed into a preface – are housed in `build/assets/`. Source files contained within document target subdirectories can generally be typeset successfully in order to produce either a finished document or a component of a finished document, such as a cover page. Asset source files, in
contrast, must always be included or otherwise combined with other source files in order to produce valid typeset-
table input.

```
build/
  11x17-landscape/........................................... A document build target directory.
  11x17-portrait/........................................... Another document build target directory.
  assets/.................................................. LaTeX files to be included into each preface layout.
    __calvino.tex
    __instrumentation.tex
    __leguin.tex
    __performance-notes.tex
  legal-landscape/.......................................... Another document build target directory.
  legal-portrait/........................................... Yet another document build target directory.
  parts.ily............................................. A LilyPond include file for generating parts.
  segments/............................................. Segment illustration LilyPond sources.
    __chemish.ily
    __cut-1.ily
    __cut-2.ily
    __komokome.ily
    __sort.ily
  segments.ily ................. A LilyPond include file giving the order of the segments to concatenate.
```

Figure 5.6: Overview of *Ersilia’s* `build/` directory.

The `build/` directory also contains two includable files sibling to the document target, `assets/` and `segments/` subdirectories. The first file, `parts.ily`, defines LilyPond commands for outputting parts. This file is discussed further in subsection 5.2.5. The second file, `segments.ily`, defines in LilyPond syntax the order in which the segment illustration sources, collected in the `segments/` subdirectory of the `build/` directory, are to be concatenated.

```
11x17-landscape/........................................... A build target directory.
  Makefile ................................................. A Makefile for GNU make affords various build tasks.
  back-cover.pdf ......................................... PDF output of the back cover LaTeX source.
  back-cover.tex ......................................... LaTeX source for the back cover.
  front-cover.pdf ......................................... PDF output of the front cover LaTeX source.
  front-cover.tex ......................................... LaTeX source for the back cover.
  music.ly ............................................... LilyPond source for the concatenated score segments.
  music.pdf ............................................... PDF output for the concatenated score segments LilyPond source.
  parts.ly............................................... LilyPond source for generating individual parts PDFs.
  preface.pdf ............................................ PDF output for the preface LaTeX source.
  preface.tex ............................................. LaTeX source for the preface.
  score.pdf ............................................... PDF output of the complete score LaTeX source.
  score.tex ............................................... LaTeX source for the complete score.
```

Figure 5.7: Directory listing of a document build target in *Ersilia*.

Each document target directory consists of a similar collection of files: LaTeX sources for the front cover, back
cover, and preface, LilyPond sources for the musical content of the score itself and parts, and a master LaTeX source which combines covers, the preface and the music into a single score PDF. The LilyPond music.ily contains include statements which pull in the segments.ily file from the build/ directory as well as the score’s main LilyPond stylesheet from the top-level stylesheets/ directory. A Makefile, which defines build commands for the GNU make command-line utility, can also help automate various document target tasks, such as recompiling all LaTeX documents in the correct order to produce a finished score.

5.1.6 The etc/ and distribution/ directories

The etc/ and distribution/ directories are perhaps the simplest. The former holds any notes, plans or to-do lists pertinent to the compositional process of the score while the latter collects completed scores and parts for each document target from the build/ directory. While note-keeping can and does happen on paper, away from the keyboard, designating a space within the score package for such miscellaneous documents allows those notes and plans to be tracked by the score package’s version control system, discussed in subsection 5.3.2.

5.1.7 The test/ directory

Finally, the test/ directory contains Python modules defining parameterized tests which verify the stability of each material and segment package. The material tests simply attempt to import and evaluate the storage format of each object defined in a material package. The classes of these objects generally have much more extensive test suites already defined either in Abjad or Consort, so simply instantiating and formatting them here suffices.

```
test/
  test_materials.py.......... Parameterized tests for validating integrity of each material package.
  test_segments.py......... Parameterized tests for validating integrity of each segment package.
```

Figure 5.8: Parameterized tests.

Segment tests attempt to both interpret each segment’s segment-maker, and also typeset the resulting LilyPond input file. Note that these tests do not attempt to guarantee that a particular segment produces some score exactly matching a target, but simply that the segment-maker manages to interpret without failure, and that LilyPond manages to typeset the resulting source without error.

5.1.8 Python packaging

Each score package should also be properly packaged according to Python standards so that it can be installed on other systems. This might strike composers as an unmotivated suggestion. Why should one structure their private
score such that it can be used by others? Making a score installable in this way affords a number of conveniences related to testing. Installable scores can be tested in virtual environments – a common Python practice – by automated test runners like tox\(^1\) or run on remote continuous-integration testing services such as Travis-CI.\(^2\)

```
ersilia/
  ├── .git/................................................................. The Git repository history.
  │   └── .gitignore................................................. File patterns to be ignored by the Git version control system.
  ├── .travis.yml ...................................................... The Travis-CI build configuration script.
  │   └── README.md. A MarkDown text file containing introductory information about the score package.
  ├── ersilia/............................................................. The score package itself.
  │   └── requirements.txt.......................................... Dependency information, for use when installing on Travis-CI.
  └── tox.ini .................................................. Configuration for the tox automated testing tool.
```

**Figure 5.9:** Overview of Ersilia’s Python packaging assets.

For those who wish to make both their scores and code completely public – as I have – installation affordances are simply necessary for letting others explore the code as quickly and easily as possible.

### 5.2 Document preparation in detail

As with working with Abjad in the simplest case, the work flow of composing and preparing scores with Consort continues to revolve around a cycle of defining musical structures in text, illustrating that music visually, then refining the textual definitions. Unlike creating simple musical examples at the command-line, as demonstrated throughout this dissertation – especially in chapter 2 –, managing a large-scale score requires considerably more tools and many more file-system assets. The following sections discuss some of the additional complexities involved in organizing and typesetting a large-scale score computationally.

#### 5.2.1 Build tools

I make use of a variety of custom build tools for managing assets within a score package which will not be discussed here in any detail as they are still rather provisional. These tools simplify various tasks such as creating new segment and material packages, executing segment definition modules in order to illustrate the contained segment-makers, comparing new illustration LilyPond sources and PDF outputs against previously rendered ones, and collecting segment LilyPond sources from each segment package into the build directory for document preparation. Simply put,

\(^1\)https://pypi.python.org/pypi/tox

\(^2\)https://travis-ci.org/
build tools for score packages streamline the problems of moving files and folders into their appropriate locations, executing Python modules, persisting code objects to disk and cleaning up after any transient files.

5.2.2 Illustrating & persisting segments

All of the examples of object illustration demonstrated throughout this dissertation involve illustrating objects in a live Python interpreter session. However, in order to construct a score made of potentially many concatenated segment illustrations, those illustrations must be persisted to disk. While persistence can certainly also be handled in a live interpreter session, by-hand illustration and persistence are both error-prone and tedious. One way to simplify the task of illustrating segments and persisting their illustrations to disk is by adding executable code to the end of each segment definition module. When run by Python as a script, those segment definition modules can be instructed to illustrate their segment makers and persist the resulting illustration and segment metadata to disk in the same directory as the segment definition module. Consider the following trivial segment definition module, complete with an executable suite at its end:

```python
import os
import consort
from abjad import persist

segment_maker = consort.SegmentMaker(
    desired_duration_in_seconds=4,
)

if __name__ == '__main__':
    illustration, metadata = segment_maker()
    directory_path = os.path.dirname(os.path.abspath(__file__))
    illustration_pdf_file_path = os.path.join(directory_path, 'illustration.pdf')
    metadata_file_path = os.path.join(directory_path, '__metadata__.py')
    persist(illustration).as_pdf(
        pdf_file_path=illustration_pdf_file_path,
        candidacy=True,
    )
    persist(metadata).as_module(
        module_file_path=metadata_file_path,
        object_name='metadata',
    )
```

When run by Python as a script – via a command like `python my_segment_definition.py` – rather than imported, the code in the segment definition module executes in the module namespace `__main__`. The conditional `if __name__ == '__main__':` guarantees then that the suite beneath that conditional executes if and only if the current module name is `__main__`,

229
again because the segment definition module was run as a script. This crucially prevents illustration code from running when the definition module is simply imported rather than executed. However, if the suite underneath the conditional does execute, it first calls the segment-maker defined earlier in that module. That segment-maker then returns both an illustration and a segment metadata dictionary. Next, file paths relative to the segment module’s file path are determined in order to persist the just-created illustration and metadata. The global variable __file__ in any Python code module gives the location of that module on the file system, from which that module’s directory can be determined. Finally, calls to Abjad’s top-level persist() function against the illustration and metadata expose persistence agent instances which afford persisting each object to disk as a LilyPond file and Python module respectively. The candidacy=True keyword argument to persist(illustration).as_pdf(...) checks whether a PDF already exists and only overwrites if the new would differ.

As mentioned in subsection 5.2.1, I make use of custom build tools for my scores which afford a more elaborate version of the illustration and persistence task outlined above. One such elaboration passes the metadata for the previous segment – if such metadata exists – to the current segment-maker’s __call__() method. Recall that the previous segment can be determined by consulting the __metadata__.py module sibling to the segment packages, as outlined in subsection 5.1.3. A modified version of the current segment-maker’s metadata is also passed as an argument at call-time, and includes both a count of the total number of segments and the current segment’s index within those segments. Such metadata allows a segment-maker to automatically determine if it is the first or last of all segments, as well as to take into account any pertinent settings effective at the end of the previous segment, such as the previous segment’s ending tempo or time signature.

Note too that persisting material illustrations can be handled in a nearly identical fashion. And unlike segments, they do not need to even optionally consult or persist metadata.

5.2.3 Collecting and concatenating segment illustrations

The model of composition afforded by Consort assumes scores consist of multiple segments, each of which is persisted as a LilyPond score context based on an identical score template. This assumption relies on LilyPond’s ability to concatenate like-named contexts. Consider the following LilyPond expression:

```lilypond
{ 
\context Score = "The Score" <<
  \context Staff = "Staff A" {c'1 d'1}
  \context Staff = "Staff B" {c'1 b1}
} >>
\context Score = "The Score" <<
```
The above example contains three like-named scores – each named “The Score” – grouped by a pair of outer braces into a single music expression. Because the scores have identical names and the staff contexts within them are also identically named – “Staff A” and “Staff B” –, LilyPond concatenates each like-named context in each score together. The resulting music expression is equivalent to the following:

\context Score = "The Score" <<
  \context Staff = "Staff A" { c'\textprime d' e' f' g' a' }
  \context Staff = "Staff B" { c'\textprime b\textprime a\textprime g\textprime f\textprime e\textprime }
>>

An identical context concatenation process is used to fuse the scores defined in each segment’s illustration file into a single music expression. First however, those segment illustrations must be collected into the segments/ directory within the build/ directory and their sources massaged to permit concatenation.

Each segment illustration, as it exists in its segment package, represents a complete, fully-typesettable LilyPond file, consisting of a LilyPond version statement, pitch-name language command, various stylesheet include commands, and that segment’s score’s context block wrapped within a \score block:

\version "2.19.17"
\language "english"

#(ly:set-option 'relative-includes #t)

\include "../../stylesheets/stylesheet.ily"

\score {
  \context Score = "The Score" <<
    ...
  >>
}

Such an input cannot simply be included along with the other segment illustration files to create a concatenated score. In fact, a construction like the following, with \score blocks nested within other \score blocks, is considered a syntax error by LilyPond:
Instead, some simple string processing must be applied against the illustration file to trim out all content besides the inner score \context block, resulting in a much thinner but includable construction.

\context Score = "The Score" <<

This trimming removes unnecessary header and styling information from each collected segment illustration and allows LilyPond to concatenate the score contexts together into a single score structure.

Segment collection then involves copying each segment illustration LilyPond source from its segment package into the segments/ subdirectory of the score package’s build/ directory, naming that copied illustration source after its originating segment package to differentiate it from the other collected segment sources, and finally trimming that source as shown above. As with segment illustration automation, I make use of some provisional build tools which simplify this process. However, a naive approach to the task of collecting and trimming segment illustration could also be implemented in a few dozen lines of Python code. With the segment illustrations collected into the build/ directory, the build/ directory’s segments.ily LilyPond include file can be updated based on the segment order encoded in the segments/ directory’s __metadata__.py module. For example, Ersilia’s segments.ily include file ultimately looks like this:

\include "segments/komokome.ly"
\include "segments/cut-1.ly"
\include "segments/sort.ly"
Finally, the updated segments.ily file can be included directly into a document target's music.ily LilyPond source file, to create the complete musical contents in the appropriate layout for that document target's paper output format. The LilyPond source for a document target's musical content, formatted for legal-size paper in portrait orientation, might then simply look like the following:

\version "2.19.17"
\language "english"

#(ly:set-option 'relative-includes #t)
\include ".//stylesheets/stylesheet.ily"
#(set-default-paper-size "legal" 'portrait)
#(set-global-staff-size 11)

\score {
  \include ".//segments.ily"
}

5.2.4 Organizing and typesetting LaTeX assets

In the spirit of LilyPond's automated musical typesetting, I have chosen to rely on LilyPond's spiritual predecessor, LaTeX, for handling all purely-textual or otherwise utilitarian typesetting in the documents I produce. LaTeX, like LilyPond, takes a textual source file as input, consisting of a variety of commands which describe the structure and content of a document, and produces a typeset target, generally as a PDF. Each document target of course requires more content than simply the musical meat of the score. Title pages, covers, prefaces and performance notes also require typesetting and should be handled as simply as possible while maintaining identical content, output formats, fonts and spacing.

The most complex LaTeX task in my document preparation process involves formatting identical textual content for different paper sizes and orientations, as with each document target's preface information. A preface containing multiple sections, each potentially containing nested lists of instruments or diagrams, may not format equally well in all desired paper dimensions. For example, the preface to *Armilla* contains six different sections of widely divergent lengths, but fits well in portrait orientations of tabloid and legal paper when split into two columns. However, when formatted on letter paper, or in landscape orientation, sections of prose no longer work in the two column layout cleanly and need to be individually resized or flowed into additional columns. Formatting
such a preface properly in both landscape and portrait orientations, in tabloid, letter and legal paper sizes requires non-trivial rearrangement and must be adjusted by hand. However, care must be taken not to duplicate content across different LaTeX source files. Duplicated prose inevitably goes out of sync, creating a tremendous cognitive burden on the composer as they prepare their documents.

This problem — needing to provide manually-tweaked alternate layouts for prose while discouraging by-hand copying — can be solved by separating out each section of text in the preface into separate includable LaTeX files, stored in the assets/ subdirectory of the build/ directory, and then including those files back into each alternative preface layout file. Such a workflow, like the use of distinct stylesheet and score files in LilyPond, separates content from layout.

Consider the preface to *Ersilia*, which contains four separate sections: two quotes, from Italo Calvino and Ursula K. Le Guin, and two passages on instrumentation and performance practice. Each of these sections is stored as a separate LaTeX file in the assets/ subdirectory of *Ersilia*’s build/ directory. When preparing prefaces for the tabloid landscape edition and the legal portrait edition, each collection of prose assets is flowed into different text grids via the textpos LaTeX package. For example, the tabloid landscape uses four separate columns, each of varying widths:

```
\begin{document}
\begin{textblock}(31)(0, 0)
  \center\huge\textbf{PREFACE}
\end{textblock}

\begin{textblock}(6)(0, 2)
  \subimport{../assets/calvino}{ }
\end{textblock}

\begin{textblock}(7)(7, 2)
  \subimport{../assets/leguin}{ }
\end{textblock}

\begin{textblock}(4)(15, 2)
  \subimport{../assets/instrumentation}{ }
\end{textblock}

\begin{textblock}(11)(20, 2)
  \subimport{../assets/performance-notes}{ }
\end{textblock}
\end{document}
```

The legal portrait version uses two columns, with the quotes on the left side and the performance instructions on the right:
In both cases, none of the actual content of the prose appears in the LaTeX files defining the layout of the prose. This guarantees that any edits or corrections to the prose will be reflected equally in all layouts of the preface.

LaTeX can also be used to join documents together. Each document target contains a master score.tex source file which contains commands for combining all other PDF components of the score together into a single PDF via LaTeX, obviating the need to use any other PDF-merging tool. For example, the 11x17 portrait score.tex for Ersilia looks like this:

```latex
\documentclass{article}
\usepackage[papersize={11in, 17in}]{geometry}
\usepackage[pdfpages]{pdfpages}
\begin{document}
\includepdf[pages=-]{ersilia-11x17-portrait-front-cover.pdf}
\includepdf[pages=-]{ersilia-11x17-portrait-preface.pdf}
\includepdf[pages=-]{ersilia-11x17-portrait-music.pdf}
\includepdf[pages=-3]{ersilia-11x17-portrait-back-cover.pdf}
\end{document}
```

5.2.5 Part extraction

When working with LilyPond and Abjad, part extraction relies on two fairly simple LilyPond mechanisms: tags and book blocks. In LilyPond, any music expression can be labeled with a tag, a string or symbol which identifies that music expression. Later, music expressions can be filtered to either remove or solely preserve any expression with a given tag. Consider the following LilyPond pseudo-code, consisting of a score containing three staves and a time signature context:
Each context contained by the Score context – the three staves and the time signature context – is tagged via a Lily-Pond \tag command, associating that context’s music expression with the tag’s symbol. The command preceding the score itself – \keepWithTag #'(time B) – indicates that the score should filter out any tagged music expression which do not belong to the list of expressions #'(time B). That is, the score should omit the two staves tagged A and C, effectively producing a score structured like so:

```
\new Score <<
  \new TimeSignatureContext = "Time Signature Context" { ... }
  \new Staff = "Staff B" { ... }
>>
```

This tagging technique is used in every score developed with Consort. Each score’s score template includes tag commands labeling both the time signature context – which must appear both in the full score and all parts as it contains time signature, tempo and rehearsal mark information – as well as all inner contexts necessary for individual performers. By constructing the appropriate \keepWithTag commands, music expressions representing each performer’s part can be constructed easily. Note the use of \tag commands throughout the structure of Armilla’s score:

```
context Score = "Armilla Score" <<
  \tag #'time
  \context TimeSignatureContext = "Time Signature Context" {
  }
  \tag #'viola-1
  \context StringPerformerGroup = "Viola 1 Performer Group" \with {
    instrumentName = \markup {
      \hcenter-in #10
      "Viola 1"
    }
    shortInstrumentName = \markup {
      \hcenter-in #10
      "Va. 1"
    }
  } <<
  \context BowingStaff = "Viola 1 Bowing Staff" {
```
The time signature context receives its own tag, and the staff groups wrapping each performer's bowing and fingering staves are also tagged appropriately.

While tagging allows for extracting parts as music expressions, it does not yet result in actual documents for each part. LilyPond's \book block structure, combined with the \bookOutputSuffix command, provide a concise mechanism for generating multiple output PDFs from a single LilyPond input file. As demonstrated throughout this document, LilyPond files are structured into blocks: context blocks, score blocks, header and paper blocks, and so forth. The highest level block is a \book block. Somewhat like parts in LaTeX, book blocks separate content from one another by page breaks. Score blocks contained in separate book blocks are guaranteed to never appear on the same page together. Furthermore, by specifying a book output suffix within each book block, LilyPond will not simply separate that book block's content by page breaks but will actually output a wholly separate PDF, whose
filename is suffixed with that book’s output suffix.

Consider this excerpt from Ersilia’s parts.ily file:

```
\book {
  \bookOutputSuffix "cello"
  \score {
    \keepWithTag #'(time cello)
    \include ".../segments.ily"
  }
}
\book {
  \bookOutputSuffix "clarinet"
  \score {
    \keepWithTag #'(time clarinet)
    \include ".../segments.ily"
  }
}
\book {
  \bookOutputSuffix "contrabass"
  \score {
    \keepWithTag #'(time contrabass)
    \include ".../segments.ily"
  }
}
```

This file contains one book block per instrument. Each book block specifies an output suffix pertinent to a specific performer in the ensemble. Each book block also contains a score block whose contents consist entirely of an include statement – pointing at segments.ily, which concatenates all score segments into a single music expression – wrapped in a \keepWithTag command which filters out everything except that performer’s musical content and the global time signature context.

With these techniques in mind, a complete parts-extraction file would look like the following, from Ersilia:

```
\version "2.19.17"
\language "english"

#(ly:set-option 'relative-includes #t)
\include ".../stylesheets/stylesheet.ily"
#(set-default-paper-size "11x17" 'landscape)
\include ".../stylesheets/parts-landscape.ily"
\include ".../parts.ily"
```

Beyond the initial LilyPond boilerplate of specifying a LilyPond version and pitch-name input language, this part extraction file simply consists of an include for the global stylesheet, a page-layout command, an include for a
stylesheet for landscape parts and finally an include for the global parts definition file, parts.ily. When interpreted by LilyPond, the above will generate one 11x17 landscape PDF per book block defined in parts.ily.

Note that there are methods by which composers can generate parts in LilyPond. Most LilyPond users who work strictly with LilyPond, writing “by hand”, would likely places each of the instrumental parts in the score into a separate file or variable. Those parts would then be combined into either the full score or a part for a single player as necessary. Because Consort produces a single score, complete with all parts joined into a single expression, filtering must be used to “strip” the score down to the desired musical elements.

5.3 Project maintenance

When one composes with code, one necessarily acts as a software developer. Therefore, scores undertaken in this fashion benefit not only from the techniques laid out earlier in this chapter, but also from those practiced daily by developers working in disciplines beyond music. Such universal techniques include version control and testing, both means of managing the stability and complexity of projects as they grow and change.

5.3.1 Automated regression testing

Regression testing examines the stability and correctness of a software system during the course of development, allowing the software’s authors to verify that changes and revisions to that system have not introduced errors or unexpected behavior. Rather than testing the system manually, software authors typically write automated regression testing batteries: collections of tests implemented as functions or classes which can be run automatically by a testing tool – a test runner –, returning the results to the authors as a report. Testing philosophies and practices in the open source community are now very diverse and sophisticated, therefore a full discussion of software testing is beyond the scope of this document. Nevertheless, I believe that testing is crucial to the development of any software system and therefore to any score or composition model implemented in code.

Both Abjad, Consort and the various scores implemented with these packages make extensive use of tests. Such tests take a variety of forms: documentation tests, unit tests, system tests and parameterized tests. Documentation testing, or “doctesting” in Python parlance, verifies that the code examples in the documentation strings accompanying classes, methods and functions are correct. Unit testing examines small fragments of code, such as individual functions, class initializers or methods, passing in a variety of input and examining the output for correctness. System testing verifies that an entire software system functions as expected. In the context of Consort, auditing the illustration produced by a fully configured segment-maker would constitute a system test as such an operation touches
upon nearly every class defined in Consort’s library. Additionally, parameterized tests provide a means of applying a single test against a variety of input. This is used extensively in Abjad and Consort to guarantee that every class in each system can be instantiated, is fully documented, can be represented as a string, can be hashed and so forth. Rather than write over 900 hundred separate tests – one for each class in Abjad – all classes can be collected in a list and passed to a single parameterized test function, which then runs itself against each class as though that were separate test.

The three scores I implemented with Consort each contain two parameterized tests in their top-level test/ directory. One test verifies that the objects defined in each material definitions are valid and contain no errors. The second parameterized test illustrates each segment definition, failing not only if the segment-maker is unable to interpret itself but also if LilyPond fails to typeset the resulting illustration source. These per-score tests serve a number of functions. They allow me to ascertain that the logic implemented in the score itself, in Consort, in Abjad and in any dependency of these projects continues to interoperate – at least non-catastrophically. They also allow me verify that the LilyPond output produced by the score’s segment-makers – along with any stylesheet information defined in the score package – is still valid LilyPond source code, and has not been deprecated by newer versions of LilyPond.

All of these types of testing combined act as a kind bulwark against backsliding both during the development of a project, in the moment, and afterward when maintaining the longevity of prior work against obsolescence.

5.3.2 Version control

Version control systems – or VCS – record changes to sets of files, tracking additions, deletions, name changes and content modifications, generally on a line-by-line basis. Changes are grouped together into bundles called commits, labeled with a timestamp and commit message elaborating on the purpose or contents of the changes. The graph of commits made against a project as recorded by the project’s VCS, where each commit is connected to some previous parent commit, constitute the history of that project. While anecdotally uncommon amongst composers, virtually all software developers make use of version control to greater or lesser degrees, for a wide variety of reasons.

Version control provides the most compact and legible way of archiving multiple versions of a project. When making changes, instead of duplicating the directory containing one’s work, giving the copied directory a name like “MyProject-year-month-day” or “MyProject-version-3” and then editing the contents of that copy, one simply works in the same directory as always, committing each change to the version control system’s history. At any point
and for any reason, any previously committed version of the project can be recalled. Reverting to an earlier version of the project does not destroy later versions which, because they have also been recorded in the project’s history, can be returned to any point. Most version control systems store their history in hidden files or directories within the project. For example, every project I work uses the VCS git, originally authored by Linus Torvalds, the chief architect of the Linux kernel. Git stores its history in a hidden directory named .git in the root of the versioned project, as indicated in subsection 5.1.8.

Many version control systems support *tagging*, the labeling of a commit in the version history as particularly important. Software projects often tag commits intended for release to the public with version numbers like 1.0, 2.5 or even v1.8.5-rc3. Likewise, important commits in the history of a score’s development might be tagged similarly. For example, the version of the project when the score was first sent to an ensemble might be tagged world-premier. Tagging allows the composer to continue revising the score while still always being able to refer back to milestones in the score’s history.

Version control systems make it easier to understand how and why a project was changed. Not only does each commit have a commit message, allowing the author to provide some explanation of their actions, but any two commits can be compared against one another as a “*diff*”, a description of the line-by-line differences between each differing file. Reading the diff between two commits is often more illuminating than reading the descriptions and intentions in the commits’ commit messages. Diffs provide a clear description of how textual content in a project has changed, making them especially useful when working in code or when composing textual input to automated typesetting programs like LilyPond and LaTeX. Consider how difficult such a version comparison task would be when comparing the contents of two project directories by hand – probably impossible, but certainly very time-consuming and error-prone.

Finally, version control affords collaboration and experimentation. Collaboration here crucially also includes collaboration with oneself. Many version control systems, including git, support *branches* which allow the revision history of the project to split into parallel time-lines rather than follow a single linear path. One can create branches whenever one desires, for example when attempting to solve a problem in more than one way where each attempt is isolated from one another but still maintained by the project’s version history. Branches can also be merged into one another, converging parallel version histories to back into a linear history. Such flexibility gives authors the freedom to embark on radical or incremental revisions and reorganizations of their projects without fear of confusion or lost

---

3http://git-scm.com/
work.
The previous chapters have discussed a computational model of music composition, implemented in the Python library Consort, and the model of notation which it extends, Abjad. The various open-source systems – \LaTeX, LilyPond, Python, etc. – which interoperate to make these twin computational models possible have also been demonstrated, and some standard solutions for establishing a document preparation workflow which streamlines and accelerates a cycle of score visualization through automated typesetting has been proposed.

As described in chapter 2, Abjad’s model of notation treats musical score as a hierarchy consisting of containers – staves, voices, measures and tuplets – and leaves – notes, rests and chords –, to which indicators – clefs, dynamics, etc. – and spanners – slurs, beams, glissandi, hairpins, and so forth – can be attached. Abjad’s model is clear and explicit whenever possible. Those objects comprising a score which a composer might wish to create – what
we might call the semantic content of the score\(^1\) – are all represented by classes in Abjad, each with a well-defined interface exposing only those properties and methods pertinent to that class. Abjad’s notation model strives for composition-process agnosticism\(^2\), allowing composers to work directly with the elemental notation objects rather than obligating them to rely on opinionated or idiosyncratic mechanisms. Abjad provides a variety of models of musical time, discussed at length in chapter 3, such as timespans and metrical hierarchies. These time models permit alternative means of constructing, coordinating and transforming musical structures than those provided by simply working with score trees directly. Timespans, especially, afford the sketching of dense polyphonic phrasing structures, and have been foundational to my working process for years, explicitly since *Aurora* and certainly with intention, although not name, for many years prior to that. Although I and the other Abjad developers have found timespans to be incredibly utilitarian, and certainly one of the most fundamental tools in our toolkit for talking about time in score, I initially\(^3\) developed them as an affordance for structuring large-scale orchestral works. All of these aspects, combined with Abjad’s tools for iterating over, selecting, and inspecting score components provides a strong foundation for others to implement their own personal models of composition: how one goes about organizing notation into a musical work.

For my part, Consort constitutes such a model of composition: a collection of high-level abstractions for organizing the elements of notation. Consort divides the process of composition into two stages – specification and interpretation – and proposes – but does not enforce – that scores be structured as a series of segments.\(^4\) Specification entails the configuration of a segment-maker – the object responsible for coordinating the creation of a segment of score – with music settings. These “settings” bundle a timespan-maker, whose responsibility it is for determining when and in which voices some material should appear, with music specifiers, objects aggregating

---

\(^1\)As opposed to those objects which are necessary or implicit, such as staff lines, bar lines, measure numbers, etc. Of course, for some composers, staff lines can and do represent semantic musical content. However, when creating input for LilyPond, I would argue that staff lines are generally simply implicit.

\(^2\)Agnosticism here stretches only so far as being agnostic of all compositional processes so long as they revolve around Western common practice notation.

\(^3\)Timespans as a compositional tool in Abjad began, in spirit, with the `timeintervaltools` subpackage, my first large contribution to Abjad, authored around 2010, which introduced a timespan-like class called `TimeInterval` and a `TimeIntervalTree` for containing them. These classes were named after the “interval-tree” data structure, often used for modeling scheduling conflicts, as it provides a highly-optimized search algorithm for finding overlap between one time interval and other time intervals or offsets. Trevor Bača later introduced a much more generalized `timespantools` subpackage and nominative `Timespan` class, into which I merged some of the more idiosyncratic `timeintervaltools` functionality, such as timespan explosion.

\(^4\)Segmentation acts both as a practical aid for typesetting, allowing smaller portions of the score to be visualized anew, and as a cognitive aid to the composer, by constraining the scope of detail they must confront during specification to a more manageable amount.
together the various makers and handlers defining a musical material. Once configured, the segment-maker may be interpreted, evaluating each of its music settings to generate a maquette – an annotated timespan structure describing the location of musical materials in the score, but not yet their notation – as well as a governing sequence of meters. That maquette is then progressively interpreted into notation, with each timespan’s annotating music specifier contributing rhythm, pitch and other typographic information to the resulting segment of score.

6.1 Concerns & Implications

As described in section 4.2, Consort’s specification and interpretation process treats the act of composition analogously to the act of compilation in software. Like a compiler, Consort’s segment-maker parses a high-level description of music – its music settings, timespan-makers and music specifiers – into an intermediate representation – the timespan maquette, itself perhaps poetically akin to computing’s notion of an abstract syntax tree\(^5\) –, and finally converts that intermediate representation into “low-level” notational primitives. In this way, Consort privileges composition with the procedural, or the general, over the specific. It is much more difficult – although not impossible – to change a single pitch at one moment in time than it is to change all of the pitches in an entire score. This has tremendous practical implications when working with such a highly-procedural system and, from experience, can be rather problematic. For example, because pitches are often “painted” onto the score in time-wise order across different voices, adding or removing a single attack-point can shift the pitches painted onto all subsequent attack-points. This expressive “entanglement” makes revising music already in the hands of performers treacherous, and I’m certainly guilty of raising some eyebrows from time to time. But the ability to describe and perform precise, mass transformations on musical materials – even if occasionally unintentionally – is one of, if not the driving motivation behind Consort. Segments may be stretched, while preserving their overall internal phrase structure. The rhythm-makers inscribing a subset of a maquette can be swapped for other rhythm-makers, yielding wholly different surface textures. Runs of notes and chords occupying weighted pitch centers can be selected and octavated en masse. Such transformations are afforded by computation. And from a computational perspective, one can consider Consort as a system which treats scores as enormous composite expressions, comprising the notational sum of the interpretation algorithm applied against each specification:

\[
\sum_{i=1}^{n} \text{Interpretation}(\text{Specification}_i) \tag{6.1}
\]

\(^5\)I do not want to overstretch this metaphor, though. I doubt it would hold up to vigorous inspection.
In considering Consort as revolving around score-as-expression, it's also worth noting that randomness – random number generators, noise functions, coin flipping, or any other such variant – plays no part in this discussion. Every segment-maker, every rhythm-maker, pitch-handler or other procedural mechanism both in Consort's ecosystem and Abjad’s, is completely deterministic. This decision is primarily pragmatic. Each package’s testing regime is considerably simpler without randomness, and the results produced by each system remain stable across multiple runs. But there’s another realisation at work here. I would argue that randomness is often a proxy for richness, detail or creative “touch”. Artists generally rely on randomness as a tool not for ideological or conceptual reasons – although some certainly do –, but simply because it affords the rapid production of material and variations on that material. In effect, a labor-saving device, and one which I relied on myself for a number of years in my acoustic music, for example in Aurora (chapter 7), as well as in all of my electronic music to this day. But there are other, less surprising means of production. Any sufficiently complex, but finite, fixed pattern of values is liable to be indistinguishable to a listener from a random sequence. This is compounded when multiple sequences of different lengths interact, as is the case in the talea timespan-makers described in subsection 3.4.2, as well as most rhythm-makers. In fact, such sequences need not be particularly long. Less than ten integers in a talea’s count sequence is often sufficient, so long as it combines with other patterns, such as the prolation-inducing extra_counts_per_division keyword. Fixed patterns also offer something that random sequences do not: the ability to both appear random, and to appear memorable. I suppose this realisation is rather trivial, but it took me a number of years to fully come to terms with it as a working philosophy and, in practice, the sequences I use for phrasing, rhythm, pitches and anything else have become shorter and shorter. In retrospect, it’s worth interrogating whether randomness is necessary or desirable at all.

Consort’s origin as a software library and the composition model it implements contains a number of other implicit assumptions. Amongst these – and I can only speak for myself – is that composition is most strongly situated in the act of specification. When I compose, I specify what will be in the score, and I specify where and how it will come to be there – that is, by what processes. This description is either a little vague or over-obvious, but consider that each of the three Invisible Cities scores, in chapter 9, chapter 10 and chapter 11, rely on only superficially different segment-makers. Virtually all of their differentiation lies in the specification of their segments, not in the process by which they are interpreted. Interpretation then becomes almost like an instrument performing these different specifications, a recapitulation of so much of my electronic composing, where custom synthesizers and audio processing networks perform different configurations over and over, auditioning for me the materials.
I will later maquette into the final work. In both cases, the acoustic and the electronic, I am also acting – quite pointedly – as the author of these “instruments” and so the algorithm used to perform a given specification is certainly not exterior to composition, just positioned differently: less specific, more general, like a compositional voice or fingerprint rather than a particular performance. Crucially, these algorithms persist from one score to the next, just as I re-use and re-combine the synthesizers I created for one electronic piece in another. The scores created with Consort are an extreme example of this re-use, undertaken specifically to investigate its practicality.

Abstraction, encapsulation, inheritance, all foundational principles in computing, point at re-use. They act to conserve labor, to persist the work done on one day in formalizing a process so that it might be reapplied on another day with little additional effort. In effect, they act as a kind of accelerant in the development and extension of further computational systems. And computing of course allows us to do things fast. But I believe it’s important to be cautious, even deeply suspicious of this. The same computing research, built over cumulative generations often in the service of finance and war, powering the so-called information age we find ourselves in today – its prizes being efficiency, productivity, accuracy, connectivity – also allows me to create new musical material faster and in greater quantity than I can ever hope to read through. This is a false economy. Unlike with electronic music, my attention simply cannot keep up with an endless stream of score. Speed has its place – and I am no more immune to its siren call than anyone else – but I hope to firmly position the benefits of computation, of library-writing elsewhere. The same qualities which afford speed often also afford structure, and ultimately extensibility. Consort and Abjad’s existence as libraries in a network of other libraries, their “library-ness”, their “open-sourceness” – even their testing regimes – act as bulwarks against the loss of knowledge encoded in them, and encourage others to interrogate, critique and extend their structure, and the music-making world they propose.

Finally, it’s important to reiterate a comment from the introduction: when working within a formal framework, one only has computational, programmatic control over what that framework describes. That which cannot be named, cannot truly be touched, and if it can be named, but cannot be described with clarity, it can only be grasped weakly, or even incorrectly. This is both cautionary to those who work in formal systems, and perhaps a panacea for those who eschew them. There is much work to be done to bring more names and descriptions into the “light,” but many things simply cannot be described specifically enough for a creature as stupid as a computer to understand.
6.2 Future work

In no way do I consider this project finished. Nor do I think a project like this – both the modeling of notation and composition – can ever be complete. There is, in my opinion, no single universal methodology to composition, nor should there ever be. And there is still quite a lot of work to do to solve an entire array of practical problems, let alone the compositional ones I often wait months or years to approach. Having devoted so much effort to large- and small-scale time structures, I need to turn my attention toward harmony and orchestration as constraints and coordinating forces. Convincing piano music remains a bugaboo. Multi-staff writing, with voices crossing between upper and lower staves and back again is well supported by LilyPond, which was designed with the spacing concerns of dense Romantic music as a foremost priority, but which requires considerable hand-adjustment. Without careful rhythmic and pitch control to account for collisions, procedurally-generated staff-changing music in a multi-voice texture quickly becomes a mess. Likewise, the use of dependent timespan-makers to create pedaling voices based on other voices, as outlined in subsection 3.4.3, sometimes produces satisfactory results, but more often doesn’t. The dependent timespan-maker is unable to truly account for the events it reacts against, and creates pedaling changes at timespan boundaries rather than at meaningful musical moments. Of course, what constitutes a meaningful musical moment is difficult to say, and I’m simply not sure yet that I can say. And in light of my cautions about naming and describing concepts computationally, I’m not sure I ever will.

Any additional affordances for idiomatic instrumental writing would be a great help: models of string instrument fingerings, and catalogues of woodwind trills and multiphonics, as well as the relative dynamic ranges in different registers and with different techniques: all of the quantitative knowledge contained in the various orchestration manuals which composers make use of. Perhaps most crucially, mechanisms for specifying that literal music expressions – not procedures to be applied during interpretation, but fully complete excerpts of music – be placed into a segment-maker’s output wherever desired would greatly extend Consort’s expressivity. The same holds true for transformations on the interpreted music, such as shifting a phrase forward or backward along the timeline, or deleting specific moments.

6.3 Parting words

My intention with providing the complete sources to both my scores and working methods – as laid out in the appendices, and discussed throughout the preceding chapters – is not that others copy me, although they certainly
can if they like. This work is open-source, after all. I wouldn’t be offended, but maybe a little disappointed. Why
would someone who managed to put together the tools and knowledge to successfully interpret one of these scores
also not take the time to place some personal stamp on their duplication by turning the knobs or mixing the potions
differently, at least a little bit? It seems so unlikely to me, and I’m sure I’ll be surprised if and when it ever happens.

Rather, my intentions are purely pedagogical. I hope this shared knowledge can be something like a lighthouse
to those who come after me, rather than hiding it away to collect dust. And while the code presented here may
become dusty, as most code does, the concepts and techniques – as separate from their concrete implementations
– most likely won’t. While I presented all of the work in this dissertation in the programming language Python
and with LilyPond as its typesetting engine, these notional and compositional models could and certainly should
be implemented in many other programming languages. A rich ecosystem of models would do wonders to keep
stagnation at bay, to share knowledge amongst colleagues, and to reduce the barriers-to-entry for newcomers to this
compositional modality.

I imagine myself, a composer ten years younger, searching for answers to many of the questions I’ve now made
good progress on solving, questions which rarely even concern making art because there is still too much ground-
work to lay. Had I a clear path then to follow, diverge from, reverse-engineer, or even wholly reject, maybe I would
have produced more music by now. Or maybe not. I no longer have the same misgivings as I did when younger
about splitting my creative energies between composition and engineering. If anything, they don’t seem that dif-
ferent to me anymore.

Basta.
Part II

Practice
This page intentionally left blank.
Aurora (2011)

A composition for string orchestra

Premièred by Ensemble Kaleidoskop
on Friday July 1st, 2011
in the Chamber Music Hall
of the Philharmonie, Berlin
Performance Notes

1 Instrumentation

mbrsi/aurora has the following instrumentation:

- 12 violins
- 4 violas
- 4 cellos
- 2 contrabasses

2 Bowing

2.1 Bow Positions

Bow positions from sul ponticello to sul tasto are indicated in 3-line tablature fragments above the main staff:

```
ordinary
sul tasto
molto sul tasto
```

```
sul ponticello
molto sul ponticello
```

2.2 Overpressure

Overpressure is indicated by a black box and bracket above the bowing-staff:

```

```

2.3 Circular Bowing / Ponticello-Tasto Tremoli

Zigzags on the bowing-staff indicate rapid circular bowing (essentially a tremolo from sul tasto to sul ponticello):

```
```

2.4 Jete / Spiccato

Dotted lines on the bowing-staff indicate a jete or similarly bounced bow:

```
```

3 Glissandi

3.1 Normal Glissandi

Two types of glissandi are prescribed. The first, with a straight line, is to be played as expected:

```
```

3.2 Oscillations

The second, with a zigzag-line, indicates a glissandi with a very, very wide vibrato, of at least a few semitones:

```
```
Contrabass /zero.oldstyle/one.oldstyle
Violin /zero.oldstyle/six.oldstyle
Violin /one.oldstyle/zero.oldstyle
Violin /one.oldstyle/two.oldstyle
Cello /zero.oldstyle/three.oldstyle
Cello /zero.oldstyle/four.oldstyle
Viola /zero.oldstyle/four.oldstyle
Cello /zero.oldstyle/one.oldstyle

69
This page intentionally left blank.
Plague Water (2014)

A composition for baritone saxophone, electric guitar, piano and percussion

Premièred by Ensemble Nikel
on Saturday April 5th, 2014
in Paine Hall, Harvard University
From Volodimir Pavliuchuk’s *Cordial Waters:*

**No.1 Plague Water** (1671, England)
- 150 gm scabious (*Scabiosa sp.*)
- 150 gm pimpernel (*Anagallis arvensis*)
- 150 gm tormentil root (*Potentilla erecta*)
- 4 litres 5% malt extract wash (strong beer as in the original)

Macerate for 12 hours and then distil.

The recommended does is a spoonful every 4 hours.

**No.2 Plague Water** (1677, England)
- 100 gm rue
- 100 gm rosemary
- 100 gm sage
- 100 gm sorrel
- 100 gm celandine (*Chelidonium majus*)
  
  (The leaves contain small amounts of toxic alkaloids which can be reduced greatly by drying the plant)
- 100 gm mugwort (*Artemisia vulgaris*)
- 100 gm bramble (blackberry) tops
- 100 gm pimpernel (*Anagallis arvensis*)
- 100 gm dragons (*Dracunculus vulgaris*)
- 100 gm agrimony (*Agrimonia eupatoria*)
- 100 gm lemonbalm
- 100 gm angelica leaves
- 4 litres white wine
  
  (substitute a 15% ABV sugar wash)

Macerate for 5 days and then distil.

---

**Baritone Saxophone**

Bartok-pizzicato indications above noteheads indicate slap tongues.

**Electric Guitar**

The electric guitar should be treated with 6 different colors, via effects pedal(s). The color to be used is indicated at the beginning of each section of the score. Pedal colors may include any combination of distortion, reverb or short delay (less than a quarter second). A volume pedal should be placed last in the effect chain, to control overall dynamic.

**Piano**

Cross-shaped noteheads indicate glissandi on the tops of the keys, without depressing the keys, played with the flesh of the fingers, or fingernails. A flat or natural sign above the glissandi determines whether to play on the black or white keys.

**Percussion**

Instrumentation is somewhat up to the discretion of the performer, but should obey the following guidelines:

- 4 wooden shakers, bamboo wind-chimes, maracas, rainsticks, cabasa, caxixi etc. These could include metal timbres, but should be primarily wood. The order of the shakers is not important. Instruments with a longer decay, and a more granular sound quality, such as rainsticks and bamboo windchimes are preferred.
- 5 wood blocks, arranged from lowest to highest. The exact pitch is not important. These could also be temple blocks. The sound quality should be very dry.
- 3 large drums, including at least one proper bass drum, arranged from lowest to highest.

Percussion should be performed with bare hands. Wooden rings may be worn to increase the overall dynamic, especially on the wood blocks. Styrofoam blocks should be placed on the bass drums, to be used during the rehearsal marks indicated in the score (4, 14, 17a, 17b). Grace notes should always be played with the hands.
Invisible Cities (i): Zaira (2014)

A composition for eight players

Premièred by Ensemble Mosaik on Saturday October 4th, 2014 in Paine Hall, Harvard University
1

In vain, great-hearted Kublai, shall I attempt to describe Zaira, city of high bastions. I could tell you how many steps make up the streets rising like stairways, and the
degree of the arcades' curves, and what kind of zinc scales cover the roofs; but I already know this would be the same as telling you nothing. The city does not consist
of this, but of relationships between the measurements of its space and the events of its past: the height of a lamppost and the distance from the ground of a
hanged usurper's swaying feet; the line strung from the lamppost to the railing opposite and the festoons that decorate the course of the queen's nuptial procession; the
height of that railing and the leap of the adulterer who climbed over it at dawn; the tilt of a guttering and a cat's progress along it as he slips into the same window; the
firing range of a gunboat which has suddenly appeared beyond the cape and the bomb that destroys the guttering; the rips in the fish net and the three old men seated on
the dock mending nets and telling each other for the hundredth time the story of the gunboat of the usurper, who some say was the queen's illegitimate son, abandoned
in his swaddling clothes there on the dock.

As this wave from memories flows in, the city soaks it up like a sponge and expands. A description of Zaira as it is today should contain all of Zaira's past. The city,
however, does not tell its past, but contains it like the lines of a hand, written in the corners of the streets, the gratings of the windows, the banisters of the steps, the
antennae of the lightning rods, the poles of the flags, every segment marked in turn with scratches, indentations, scrolls.

- Italo Calvino, *Invisible Cities*

2 Instrumentation

- Flute, with brazil nut shaker
- Oboe
- Clarinet in b-flat, with brazil nut shaker
- Percussion

<table>
<thead>
<tr>
<th>Metals</th>
<th>tam-tam</th>
<th>tom cymbal</th>
<th>middle cymbal</th>
<th>high cymbal</th>
<th>brake drum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Woods</td>
<td>bamboo windchimes</td>
<td>guero</td>
<td>tambourine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drums</td>
<td>bass drum</td>
<td>kick drum</td>
<td>low tom</td>
<td>high tom</td>
<td></td>
</tr>
</tbody>
</table>

Mallets: hard sticks or bare hands, wire brushes, superballs

- Piano

Prepare the lowest and highest octaves with any combination of felt, tape or rubber to dampen and distort the timbre of the strings.

Guero passages should be played with a piece of hard paper or plastic, on the keys. The register of the motions is left to the performer.

- Violin, with brazil nut shaker
- Viola, with brazil nut shaker
- Cello
Invisible Cities (i):
ZAIRA
for Ensemble Mosaik

Josiah Wolf Oberholtzer /one.oldstyle/nine.oldstyle/eight.oldstyle/four.oldstyle
This page intentionally left blank.

A composition for viola duet

Premièred by Elizabeth Weisser &
John Pickford Richards
on Saturday February 7th, 2015
in Paine Hall, Harvard University
PREFACE

Whether Armilla is like this because it is unfinished or because it has been demolished, whether the cause is some enchantment or only a whim, I do not know. The fact remains that it has no walls, no ceilings, no floors: it has nothing that makes it seem a city except the water pipes that rise vertically where the houses should be and spread out horizontally where the floors should be: a forest of pipes that end in taps, showers, spouts, overflows. Against the sky a lavabo’s white stands out, or a bathtub, or some other porcelain, like late fruit still hanging from the boughs. You would think that the plumbers had finished their job and gone away before the bricklayers arrived; or else their hydraulic systems, indestructible, had survived a catastrophe, an earthquake, or the corrosion of termites.

Abandoned before or after it was inhabited, Armilla cannot be called deserted. At any hour, raising your eyes among the pipes, you are likely to glimpse a young woman, or many young women, slender, not tall of stature, luxuriating in the bathtubs or arching their backs under the showers suspended in the void, washing or drying or perfuming themselves, or combing their long hair at a mirror. In the sun, the threads of water fanning from the showers glister, the jets of the taps, the sprays, the spouts, the sponges’ suds.

I have come to this explanation: the streams of water channeled in the pipes of Armilla have remained in the possession of nymphs and naiads. Accustomed to traveling along underground veins, they found it easy to enter the new aquatic realm, to burst from multiple fountains, to find new mirrors, new games, new ways of enjoying the water. Their invasion may have driven out the human beings, or Armilla may have been built by humans as a votive offering to win the favor of the nymphs, offended at the misuse of the waters. In any case, now they seem content, these maidens: in the morning you hear them singing.

- Italo Calvino, Invisible Cities

2

The dunes ran inland, low and grassy, for half a mile or so, and then there were lagoons, thick with sedge and saltreeds, and beyond those, low hills lay yellow-brown and empty out of sight. Beautiful and desolate was Selidor. Nowhere on it was there any mark of man, his work or habitation. There were no beasts to be seen, and the reed-filled lakes bore no flocks of gulls or wild geese or any bird.

- Ursula Le Guin, The Farthest Shore

3 Bow contact points

The current position along the bow as it contacts the strings is indicated with fractions, where 0/1 indicates the frog, and 1/1 indicates the tip of the bow. Continuous bowing is shown by lines connecting bow contact fractions.

4 Dynamics

Dynamics are always in terms of effort, not effect. When bowing very quickly and with strong dynamic, the effect should be a traditional forte. Likewise, when bowing slowly and with a light dynamic, the effect should be a traditional piano. Slow bowing with strong dynamic should result in various colors of scratch, while fast bowing with light pressure should give various qualities of flautando (depending of course on where on the string the bow is contacting).

5 String contact points

D.P. Dietro ponticello: behind the bridge.
M.S.P. Molto sul ponticello
S.P. Sul ponticello
Ord. Ordinario
S.T. Sul tasto
M.S.T. Molto sul tasto

When bowing behind the bridge, the fingering staff switches to percussion clef. The behind-the-bridge string to bow on are given by the four spaces of the five-line-staff, with string IV being the lowest space and I the highest.

6 Other techniques

Across-the-string tremoli
Indicated by traditional tremolo hashes on the bow tablature’s rhythm staff

Along-the-string tremoli
Indicated by zigzag bow tablature glissandi

Thrown bow
Indicated by dashed bow tablature glissandi

Pizzicati
Indicated with a cross notehead in the tablature staff.

Accents
Accents in the bow tablature staff indicate a sudden staccato increase in bow pressure.

Tremoli, both across- and along-the-string, should be very tight. When the two techniques appear simultaneously, the resulting motion is tightly circular bowing.
Invisible Cities (i):

ARMILLA
(a botanical survey of the uninhabited southern isles)

for Elizabeth Weisser and John Pickford Richards

Josiah Wolf Oberholtzer (1986)
This page intentionally left blank.

A composition for chamber orchestra

Premièred by Ensemble Dal Niente
on Saturday May 16th, 2015
in Paine Hall, Harvard University
1

In Ersilia, to establish the relationships that sustain the city’s life, the inhabitants stretch strings from the corners of the houses, white or black or gray or black-and-white according to whether they mark a relationship of blood, of trade, authority, agency. When the strings become so numerous that you can no longer pass among them, the inhabitants leave: the houses are dismantled, only the strings and their supports remain.

From a mountainside, camping with their household goods, Ersilia’s refugees look at the labyrinth of taut strings and poles that rise in the plain. That is the city of Ersilia still, and they are nothing.

They rebuild Ersilia elsewhere. They weave a similar pattern of strings which they would like to be more complex and at the same time more regular than the other. Then they abandon it and take themselves and their houses still farther away.

Thus, when traveling in the territory of Ersilia, you come upon the ruins of abandoned cities, without the walls which do not last, without the bones of the dead which the wind rolls away: spiderwebs of intricate relationships seeking a form.

- Italo Calvino, Invisible Cities

2

"Are we still in the South Reach?"

"Reach? No. The islands – "The chief moved his slender black hand in an arc, no more than a quarter of the compass, north to east. "The islands are there," he said. "All the islands." Then showing all the evening sea before them, from north through west to south, he said, "The sea."

"What land are you from, lord?"

"No land. We are the Children of the Open Sea."

Arren looked at his keen face. He looked about him at the great raft with its temple and its tall idols, each carved from a single tree, great god-figures mixed of dolphin, fish, man, and seabird; at the people busy at their work, weaving, carving, fishing, cooking on raised platforms, tending babies; at the other rafts, seventy at least, scattered out over the water in a great circle perhaps a mile across. It was a town: smokes rising in thin wisps from distant houses, the voices of children high on the wind. It was a town, and under its floors was the abyss.

- Ursula LeGuin, The Farthest Shore

3 Instrumentation

- Flute
- Bass clarinet
- Oboe
- Baritone saxophone
- Acoustic guitar
- Piano
- Percussion
  - bamboo wind chimes
  - four toms
  - five wood blocks
  - snare drum
  - marimba
  - crotales, two octaves
  - tam-tam
  - bass drum
- Violin
- Viola
- Cello
- Contrabass

4 Performance notes

Six players – flute, bass clarinet, oboe, violin, viola and cello – receive a shaker – caxixi, maraca or similar. Four players – guitar, piano, percussion and contrabass – receive a chromatic pitch pipe – a circular harmonic-like instrument generally used for tuning vocal groups.

The shakers should be placed on or suspended from their respective performer’s music stands, or wherever convenient.

The pitch-pipes should be played by inhaling or exhaling – as indicated – through fully half of the circumference of the instrument, creating a rich cluster.

For all winds, a + -symbol indicates slap tonguing.

For guitar, the coda symbol indicates a percussive damping of the strings.

Piano plays with pedal to their discretion throughout the first four sections of the piece. The sound should be generally dry, although some pedal should be used when appropriate for phrasing and blending, especially on tremolo passages. The sustain pedal should remain fully depressed for the entirety of section D. Inside-piano glissandos are notated proportional to the lower interior portion of the instrument – from the lowest string up to the first cross-bar – and should be played with the fingertip.

Percussion should use hard sticks on toms, woodblocks, snares and crotales, and softer mallets on marimba, bass drum and tam-tam (when ergonomic to do so).

All mordents and trills are a major 2nd, unless otherwise specified. All tremolos are unmeasured and should evoke an even, cloud-like texture.
Invisible Cities (iii):
ERSILIA
(a botanical survey of the uninhabited northeastern isles)
for Ensemble Dal Niente

Josiah Wolf Oberholtzer (1984)
Part III

Appendices
This page intentionally left blank.
A.1 consort.tools.AbsolutePitchHandler

```python
# -*- encoding: utf-8 -*-
from __future__ import print_function
from consort.tools.PitchHandler import PitchHandler

class AbsolutePitchHandler(PitchHandler):
    r'''Absolute pitch maker.
    ::

        >>> import consort
        >>> pitch_handler = consort.AbsolutePitchHandler(
        ...     pitch_specifier="c' d' e' f'",
        ... )
        >>> print(format(pitch_handler))
        consort.tools.AbsolutePitchHandler(
            pitch_specifier=consort.tools.PitchSpecifier(
                pitch_segments=(
                    pitchtools.NamedPitch("c'"),
                    pitchtools.NamedPitch("d'"),
                    pitchtools.NamedPitch("e'"),
                    pitchtools.NamedPitch("f'"),
                ),
                item_class=pitchtools.NamedPitch,
            ),
            ratio=mathtools.Ratio((1,)),
        ),
```
### CLASS VARIABLES ###

__slots__ = ()

### SPECIAL METHODS ###

```python
def __call__(self, attack_point_signature, logical_tie, music_specifier, pitch_choices, previous_pitch, seed_session):
    pitch = self._get_pitch(
        attack_point_signature,
        pitch_choices,
        previous_pitch,
        seed_session.current_phrased_voicewise_logical_tie_seed,
    )
    pitch = self._apply_deviation(
        pitch,
        seed_session.current_unphrased_voicewise_logical_tie_seed,
    )
    return pitch
```

### PRIVATE METHODS ###

```python
def _get_pitch(self, attack_point_signature, pitch_choices, previous_pitch, seed):
    pitch = pitch_choices[seed]
    if pitch_choices and 1 < len(set(pitch_choices)) and self.forbid_repetitions:
        if self.pitch_application_rate == 'phrase':
            if attack_point_signature.is_first_of_phrase:
                while pitch == previous_pitch:
                    seed += 1
                    pitch = pitch_choices[seed]
        elif self.pitch_application_rate == 'division':
            if attack_point_signature.is_first_of_division:
                while pitch == previous_pitch:
                    seed += 1
                    pitch = pitch_choices[seed]
```
else:
    while pitch == previous_pitch:
        seed += 1
        pitch = pitch_choices[seed]
    return pitch

A.2 consort.tools.AfterGraceSelectorCallback

### CLASS VARIABLES ###
__slots__ = ()

### SPECIAL METHODS ###
def __call__(self, expr):
    assert isinstance(expr, tuple), repr(tuple)
    result = []
    for subexpr in expr:
        subresult = []
        for x in subexpr:
            subresult.append(x)
            if inspect_(x).get_grace_containers('after'):
                subresult = selectiontools.Selection(subresult)
                result.append(subresult)
        subresult = []
        if subresult:
            subresult = selectiontools.Selection(subresult)
            result.append(subresult)
    return tuple(result)

A.3 consort.tools.annotate

def make_annotated_phrase(phrase, color=None):
    duration = inspect_(phrase).get_duration()
annotated_phrase = scoretools.FixedDurationTuplet(duration)

durations = [inspect_(_,).get_duration() for _ in phrase]
leaves = scoretools.make_leaves([], durations)
annotated_phrase.extend(leaves)

if color:
    override(annotated_phrase).tuplet_bracket.color = color
return annotated_phrase


def make_annotated_division(division, color=None):
    duration = inspect_(division).get_duration()
    if duration != 1:
        note = scoretools.Note("c'''1")
        annotated_division = scoretools.Tuplet(duration, (note,))
    else:
        note = scoretools.Note("c'''2")
        annotated_division = scoretools.Tuplet(2, (note,))
    leaves = division.select_leaves()
    prototype = (scoretools.Rest, scoretools.MultimeasureRest)
    if all(isinstance(_, prototype) for _ in leaves):
        manager = override(annotated_division)
        manager.tuplet_bracket.dash_fraction = 0.1
        manager.tuplet_bracket.dash_period = 1.5
        manager.tuplet_bracket.style = \\
        schemetools.SchemeSymbol('dashed-line')
    if color:
        override(annotated_division).tuplet_bracket.color = color
    return annotated_division


def make_empty_phrase_divisions(phrase):
    inner_container = scoretools.Container()
    outer_container = scoretools.Container()
    for division in phrase:
        duration = inspect_(division).get_duration()
        multiplier = durationtools.Multiplier(duration)
        inner_skip = scoretools.Skip(1)
        outer_skip = scoretools.Skip(1)
        attach(multiplier, inner_skip)
        attach(multiplier, outer_skip)
        inner_container.append(inner_skip)
        outer_container.append(outer_skip)
    return inner_container, outer_container


def annotate(context, nonsilence=None):
    prototype = consort.MusicSpecifier
    silenceSpecifier = consort.MusicSpecifier()
    annotated_context = context
    context_mapping = {}
    for voice in iterate(annotated_context).by_class(scoretools.Voice):
        if voice.context_name == 'Dynamics':
            continue
        division_voice = scoretools.Context(

context_name='AnnotatedDivisionsVoice',
)

phrase_voice = scoretools.Context(
    context_name='AnnotatedPhrasesVoice',
)

for phrase in voice:
    color = None
    if nonsilence:
        music_specifier = inspect_(phrase).get_indicator(prototype)
        if music_specifier == silence_specifier:
            inner_container, outer_container = \
            make_empty_phrase_divisions(phrase)
            division_voice.append(inner_container)
            phrase_voice.append(outer_container)
            continue
        if music_specifier:
            color = music_specifier.color
        for division in phrase:
            annotation_division = make_annotated_division(division, color)
            division_voice.append(annotation_division)
            annotation_phrase = make_annotated_phrase(phrase, color)
            phrase_voice.append(annotation_phrase)

parent = inspect_(voice).get_parentage().parent
if parent not in context_mapping:
    context_mapping[parent] = []
context_mapping[parent].append(division_voice)
context_mapping[parent].append(phrase_voice)

for context, annotation_voices in context_mapping.items():
    context.is_simultaneous = True
    context.extend(annotation_voices)

A.4  consort.tools.AttachmentExpression

# -*- encoding: utf-8 -*-
from __future__ import print_function
import collections
import copy
import funcsigs
from abjad import attach
from abjad import new
from abjad.tools import datastructuretools
from abjad.tools import scoretools
from abjad.tools import selectortools
from abjad.tools import sequencetools
from abjad.tools import spannertools
from consort.tools.HashCachingObject import HashCachingObject

class AttachmentExpression(HashCachingObject):
    r'''An attachment specifier.
    .. container:: example

        ::"
>>> import consort
>>> attachment_expression = consort.AttachmentExpression(
...     attachments=(indicatortools.Articulation('>'),),
...     selector=selectortools.Selector().by_leaves().by_run(Note)[0],
... )
>>> print(format(attachment_expression))
consort.tools.AttachmentExpression(
    attachments=datastructuretools.TypedList(
        [indicatortools.Articulation('>'),
    ],
    selector=selectortools.Selector(
        callbacks=(
            selectortools.PrototypeSelectorCallback(
                prototype=scoretools.Leaf,
            ),
            selectortools.RunSelectorCallback(
                prototype=scoretools.Note,
            ),
            selectortools.ItemSelectorCallback(
                item=0,
                apply_to_each=True,
            ),
        ),
    ),
)
::

>>> attachment_expression = consort.AttachmentExpression(
...     attachments=(
...         consort.SimpleDynamicExpression(
...             hairpin_start_token='sfz',
...             hairpin_stop_token='o',
...         ),
...     ),
...     selector=selectortools.Selector().by_leaves().by_run(Note),
... )
::

>>> staff = Staff("c'8 r8 d'8 e'8 r8 f'8 g'8 a'8")
>>> attachment_expression(staff)
>>> print(format(staff))
\new Staff { 
c'8 \sfz  
r8  
\override Hairpin #'circled-tip = ##t  
d'8 \> \sfz  
e'8 \!  
\revert Hairpin #'circled-tip  
r8
.. container:: example

    >>> attachment_expression = consort.AttachmentExpression(
        ...    attachments=spannertools.Slur(),
        ...    )
    >>> staff = Staff("c'4 d'4 e'4 f'4")
    >>> attachment_expression(staff)
    >>> print(format(staff))
    \new Staff {
        c'4 (
        d'4
        e'4
        f'4 )
    }

.. container:: example

    >>> staff = Staff("c'4 d'4 e'4 f'4")
    >>> attachment_expression = consort.AttachmentExpression(
        ...    attachments=[
        ...        indicatortools.Articulation('accent'),
        ...        indicatortools.Articulation('staccato'),
        ...    ],
        ...    selector=selectortools.Selector().by_logical_tie()[0]
        ...    )
    >>> attachment_expression(staff)
    >>> print(format(staff))
    \new Staff {
        c'4 -\accent -\staccato
        d'4 -\accent -\staccato
        e'4 -\accent -\staccato
        f'4 -\accent -\staccato
    }

### CLASS VARIABLES ###

```python
__slots__ = ('_attachments',
            '_scope',
```
```python
130  
131     '_selector',
132     '_is_annotation',
133     '_is_destructive',
134  )
135
136  ### INITIALIZER ###
137  def __init__(self,
138      attachments=None,
139      selector=None,
140      scope=None,
141      is_annotation=None,
142      is_destructive=None,
143  ):
144      HashCachingObject.__init__(self)
145      if attachments is not None:
146          if not isinstance(attachments, collections.Sequence):
147              attachments = (attachments,)
148          attachments = datastructuretools.TypedList(attachments)
149          self._attachments = attachments
150      if selector is not None:
151          assert isinstance(selector, selectortools.Selector)
152          self._selector = selector
153      if scope is not None:
154          if isinstance(scope, type):
155              assert issubclass(scope, scoretools.Component)
156          else:
157              assert isinstance(scope, (scoretools.Component, str))
158          self._scope = scope
159      if is_annotation is not None:
160          is_annotation = bool(is_annotation)
161          self._is_annotation = is_annotation
162      if is_destructive is not None:
163          is_destructive = bool(is_destructive)
164          self._is_destructive = is_destructive
165
166  ### PUBLIC METHODS ###
167  def __call__(self,
168      music,
169      name=None,
170      rotation=0,
171  ):
172      selector = self.selector
173      if selector is None:
174          selector = selectortools.Selector()
175      selections = selector(music, rotation=rotation)
176      self._apply_attachments(  
177          selections,
178          name=name,
179          rotation=rotation,
180      )
```
### PRIVATE METHODS ###

def _apply_attachments(self, selections, name=None, rotation=None):
    if not self.attachments:
        return
    all_attachments = datastructuretools.CyclicTuple(self.attachments)
    for i, selection in enumerate(selections, rotation):
        attachments = all_attachments[i]
        # print(i, selection, attachments)
        if attachments is None:
            continue
        if not isinstance(attachments, collections.Sequence):
            attachments = (attachments,)
        for attachment in attachments:
            # print('| ' + repr(attachment))
            # spanners
            if isinstance(attachment, spannertools.Spanner):
                attachment = copy.copy(attachment)
                attach(attachment, selection, name=name)
            elif isinstance(attachment, type) and \
                issubclass(attachment, spannertools.Spanner):
                attachment = attachment()
                attach(attachment, selection, name=name)
            # expressions
            elif hasattr(attachment, '__call__'):
                signature = funcsignatures.signature(attachment.__call__)
                if 'seed' in signature.parameters:
                    attachment(selection, seed=rotation, name=name)
                elif 'rotation' in signature.parameters:
                    attachment(selection, rotation=rotation, name=name)
                elif 'name' in signature.parameters:
                    attachment(selection, name=name)
                else:
                    attachment(selection)
            # indicators
            else:
                if isinstance(selection, scoretools.Leaf):
                    selection = (selection,)
                for component in selection:
                    attachment = copy.copy(attachment)
                    attach(
                        attachment,
                        component,
                        name=name,
                        scope=self.scope,
                        is_annotation=self.is_annotation,
                    )

### PUBLIC METHODS ###

def reverse(self):
    attachments = sequencetools.Sequence(*self.attachments)
    return new(self,
    attachments=attachments.reverse(),
)

def rotate(self, n=0):
    attachments = sequencetools.Sequence(*self.attachments)
    return new(self,
        attachments=attachments.rotate(n),
    )

### PUBLIC PROPERTIES ###

@property
def attachments(self):
    return self._attachments

@property
def is_annotation(self):
    return self._is_annotation

@property
def is_destructive(self):
    return self._is_destructive

@property
def scope(self):
    return self._scope

@property
def selector(self):
    return self._selector

A.5  consort.tools.AttachmentHandler

# -*- encoding: utf-8 -*-
from __future__ import print_function
import collections
from abjad import inspect_
from abjad import iterate
from abjad.tools import abctools
from abjad.tools import scoretools
from abjad.tools import selectortools
from abjad.tools import systemtools

class AttachmentHandler(abctools.AbjadValueObject):
    r'''An attachment maker.''

    ::

        >>> import consort
        >>> attachment_handler = consort.AttachmentHandler()
        >>> print(format(attachment_handler))
consort.tools.AttachmentHandler()
```python

### CLASS VARIABLES ###

```slots__ = ('_attachment_expressions',)

### INITIALIZER ###

def __init__(self, **attachment_expressions):
    import consort
    prototype = consort.AttachmentExpression
    validated_attachment_expressions = {}
    for name, attachment_expression in attachment_expressions.items():
        if attachment_expression is None:
            continue
        if not isinstance(attachment_expression, prototype):
            attachment_expression = consort.AttachmentExpression(
                attachments=attachment_expression,
            )
        validated_attachment_expressions[name] = attachment_expression
    self._attachment_expressions = validated_attachment_expressions

### SPECIAL METHODS ###

def __call__(self, music, seed=0,):
    assert isinstance(music, scoretools.Container)
    if not self.attachment_expressions:
        return
    items = self.attachment_expressions.items()
    #print('
nondestructive...')
    destructive_expressions = set()
    selectors = set()
    selectors_to_expressions = {}
    for item in items:
        name, attachment_expression = item
        if attachment_expression.is_destructive:
            destructive_expressions.add(item)
            continue
        selector = attachment_expression.selector
        if selector is None:
            selector = selectortools.Selector()
        selectors.add(selector)
        if selector not in selectors_to_expressions:
```
selectors_to_expressions[selector] = set()
selectors_to_expressions[selector].add(item)

selectors_to_selections = selectortools.Selector.run_selectors(
music, selectors,
rotation=seed,
)

for selector in selectors:
    expressions = selectors_to_expressions[selector]
    selections = selectors_to_selections[selector]
    for name, attachment_expression in expressions:
        attachment_expression._apply_attachments(
            selections,
            name=name,
            rotation=seed,
        )

    #print('\t...done')
if destructive_expressions:
    #print('destructive...')
    for name, attachment_expression in destructive_expressions:
        #print('\t', name)
        attachment_expression(
            music,
            name=name,
            rotation=seed,
        )
    #print('\t...done')

def __getattr__(self, item):
    if item in self.attachment_expressions:
        return self.attachment_expressions[item]
    return object.__getattribute__(self, item)

### PRIVATE METHODS ###

@staticmethod
def _process_session(segment_maker, verbose=None):
    import consort
    score = segment_maker.score
    counter = collections.Counter()
    for voice in iterate(score).by_class(scoretools.Voice):
        progress_indicator = systemtools.ProgressIndicator(
            message=' decorating {}'.format(voice.name),
            verbose=verbose,
        )
        with progress_indicator:
            for container in voice:
                prototype = consort.MusicSpecifier
                music_specifier = inspect_(container).get_effective(prototype)
                maker = music_specifier.attachment_handler
                if maker is None:
                    continue
                if music_specifier not in counter:
                    seed = music_specifier.seed or 0
counter[music_specifier] = seed
seed = counter[music_specifier]
    maker(container, seed=seed)
counter[music_specifier] -= 1
progress_indicator.advance()

### PRIVATE PROPERTIES ###

@property
def _storage_format_specification(self):
    return systemtools.StorageFormatSpecification(
        self,
        keyword_argument_names=sorted(self.attachment_expressions.keys()),
    )

### PUBLIC PROPERTIES ###

@property
def attachment_expressions(self):
    return self._attachment_expressions.copy()

A.6 consort.tools.AttackPointSignature

```python
# -*- encoding: utf-8 -*-
from abjad import inspect_
from abjad import iterate
from abjad.tools import abctools
from abjad.tools import durationtools
from abjad.tools import scoretools

class AttackPointSignature(abctools.AbjadValueObject):
    r'''An attack point signature.''

    ::

        >>> import consort
        >>> attack_point_signature = consort.AttackPointSignature(
        ... division_position=0,
        ... phrase_position=(1, 2),
        ... segment_position=(4, 5),
        ... )
        >>> print(format(attack_point_signature))
        consort.tools.AttackPointSignature(
            division_index=0,
            division_position=durationtools.Multiplier(0, 1),
            logical_tie_index=0,
            phrase_position=durationtools.Multiplier(1, 2),
            segment_position=durationtools.Multiplier(4, 5),
            total_divisions_in_phrase=1,
            total_logical_ties_in_division=1,
        )
```

...
### CLASS VARIABLES ###

```python
__slots__ = ('_division_index',
            '_division_position',
            '_logical_tie_index',
            '_phrase_position',
            '_segment_position',
            '_total_divisions_in_phrase',
            '_total_logical_ties_in_division',
```  

### INITIALIZER ###

```python
def __init__(
    self,
    division_index=0,
    division_position=0,
    logical_tie_index=0,
    phrase_position=0,
    segment_position=0,
    total_divisions_in_phrase=1,
    total_logical_ties_in_division=1,
):
    division_index = int(division_index)
    division_position = durationtools.Multiplier(division_position)
    logical_tie_index = int(logical_tie_index)
    phrase_position = durationtools.Multiplier(phrase_position)
    segment_position = durationtools.Multiplier(segment_position)
    total_divisions_in_phrase = int(total_divisions_in_phrase)
    total_logical_ties_in_division = int(total_logical_ties_in_division)
    assert 0 <= logical_tie_index < total_logical_ties_in_division
    assert 0 <= division_index < total_divisions_in_phrase
    assert 0 <= division_position <= 1
    assert 0 <= phrase_position <= 1
    assert 0 <= segment_position <= 1
    self._division_index = division_index
    self._division_position = division_position
    self._logical_tie_index = logical_tie_index
    self._phrase_position = phrase_position
    self._segment_position = segment_position
    self._total_divisions_in_phrase = total_divisions_in_phrase
    self._total_logical_ties_in_division = 
    total_logical_ties_in_division
```

### PRIVATE METHODS ###

```python
@staticmethod
def _find_position(
    logical_tie_start_offset,
    bounding_start_offset,
    bounding_stop_offset,
):
```
duration = bounding_stop_offset - bounding_start_offset
start_offset = logical_tie_start_offset - bounding_start_offset
if duration == 0:
    return durationtools.Multiplier(0)
position = start_offset / duration
assert 0 <= position <= 1
return position

### PUBLIC METHODS ###

@classmethod
def from_logical_tie(cls, logical_tie):
    import consort
    logical_tie_start_offset = logical_tie.get_timespan().start_offset
    phrase = consort.SegmentMaker.logical_tie_to_phrase(logical_tie)
    phrase_logical_ties = cls._collect_logical_ties(phrase)
    phrase_position = cls._find_position(
        logical_tie_start_offset,
        phrase_logical_ties[0].get_timespan().start_offset,
        phrase_logical_ties[-1].get_timespan().start_offset,
    )
    division = consort.SegmentMaker.logical_tie_to_division(logical_tie)
    division_index = phrase.index(division)
    total_divisions_in_phrase = len(phrase)
    division_logical_ties = cls._collect_logical_ties(division)
    division_position = cls._find_position(
        logical_tie_start_offset,
        division_logical_ties[0].get_timespan().start_offset,
        division_logical_ties[-1].get_timespan().start_offset,
    )
    logical_tie_index = division_logical_ties.index(logical_tie)
    total_logical_ties_in_division = len(division_logical_ties)
    voice = consort.SegmentMaker.logical_tie_to_voice(logical_tie)
    segment_timespan = inspect_(voice).get_timespan()
    segment_position = cls._find_position(
        logical_tie_start_offset,
        segment_timespan.start_offset,
        segment_timespan.stop_offset,
    )
    signature = cls(
        division_index=division_index,
        division_position=division_position,
        logical_tie_index=logical_tie_index,
        phrase_position=phrase_position,
        segment_position=segment_position,
        total_divisions_in_phrase=total_divisions_in_phrase,
        total_logical_ties_in_division=total_logical_ties_in_division,
    )
    return signature
### PRIVATE METHODS ###

```python
@staticmethod
def _collect_logical_ties(container):
    logical_ties = []
    for leaf in iterate(container).by_class(scoretools.Note):
        leaf_logical_tie = inspect_(leaf).get_logical_tie()
        if leaf is not leaf_logical_tie.head:
            continue
        logical_ties.append(leaf_logical_tie)
    return logical_ties
```

### PUBLIC PROPERTIES ###

```python
@property
def division_index(self):
    return self._division_index

@property
def division_position(self):
    return self._division_position

@property
def is_first_of_division(self):
    if not self.logical_tie_index:
        return True
    return False

@property
def is_first_of_phrase(self):
    if not self.logical_tie_index and not self.division_index:
        return True
    return False

@property
def logical_tie_index(self):
    return self._logical_tie_index

@property
def phrase_position(self):
    return self._phrase_position

@property
def segment_position(self):
    return self._segment_position

@property
def total_divisions_in_phrase(self):
    return self._total_divisions_in_phrase

@property
def total_logical_ties_in_division(self):
    return self._total_logical_ties_in_division
```
class BoundaryTimespanMaker(TimespanMaker):

    r'''A boundary timespan-maker.
    
    >>> import consort
    >>> timespan_maker = consort.BoundaryTimespanMaker(
    ...     start_talea=rhythmmakertools.Talea(
    ...         counts=[1],
    ...         denominator=2,
    ...     ),
    ...     stop_talea=rhythmmakertools.Talea(
    ...         counts=[1],
    ...         denominator=4,
    ...     ),
    ...     voice_names=('Violin 1 Voice', 'Violin 2 Voice'),
    ... )
    >>> print(format(timespan_maker))
    consort.tools.BoundaryTimespanMaker(
        start_talea=rhythmmakertools.Talea(
            counts=(1,),
            denominator=2,
        ),
        stop_talea=rhythmmakertools.Talea(
            counts=(1,),
            denominator=4,
        ),
        voice_names=('Violin 1 Voice', 'Violin 2 Voice'),
    )
    
    >>> timespan_inventory = timespantools.TimespanInventory([
    ...     consort.PerformedTimespan(
    ...         start_offset=0,
    ...         stop_offset=1,
    ...         voice_name='Violin 1 Voice',
    ...     ),
    ...     consort.PerformedTimespan(
    ...         start_offset=(1, 2),
    ...         stop_offset=(3, 2),
    ...         voice_name='Violin 2 Voice',
    ...     ),
    ...     consort.PerformedTimespan(
    ...         start_offset=-3,
    ...         stop_offset=-1,
    ...         voice_name='Violin 2 Voice',
    ...     ),
    ... ])
    >>> print(format(timespan_inventory))
    consort.tools.PerformedTimespanInventory([
        consort.PerformedTimespan(
            start_offset=0,
            stop_offset=1,
            voice_name='Violin 1 Voice',
        ),
        consort.PerformedTimespan(
            start_offset=(1, 2),
            stop_offset=(3, 2),
            voice_name='Violin 2 Voice',
        ),
        consort.PerformedTimespan(
            start_offset=-3,
            stop_offset=-1,
            voice_name='Violin 2 Voice',
        ),
    ])

    >>> print(format(timespan_inventory))
... start_offset=3,
... stop_offset=4,
... voice_name='Violin 2 Voice',
... ),
... ])

::

>>> music_specifiers = {'Cello Voice': None}
>>> target_timespan = timespantools.Timespan(0, 10)
>>> timespan_inventory = timespan_maker(
... music_specifiers=music_specifiers,
... target_timespan=target_timespan,
... timespan_inventory=timespan_inventory,
... )
>>> print(format(timespan_inventory))
timespantools.TimespanInventory(
[  
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(1, 2),
        voice_name='Cello Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(1, 1),
        voice_name='Violin 1 Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(1, 2),
        stop_offset=durationtools.Offset(3, 2),
        voice_name='Violin 2 Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(3, 2),
        stop_offset=durationtools.Offset(7, 4),
        voice_name='Cello Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(4, 1),
        voice_name='Cello Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(3, 1),
        stop_offset=durationtools.Offset(4, 1),
        voice_name='Violin 2 Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(4, 1),
        stop_offset=durationtools.Offset(17, 4),
        voice_name='Cello Voice',
    ),
]  

### CLASS VARIABLES ###

```python
__slots__ = ('_labels',
            '_start_talea',
            '_start_groupings',
            '_stop_talea',
            '_stop_groupings',
            '_voice_names',
           )
```

### INITIALIZER ###

```python
def __init__(
    self,
    start_talea=None,
    stop_talea=None,
    start_groupings=None,
    stop_groupings=None,
    labels=None,
    output_masks=None,
    padding=None,
    seed=None,
    timespan_specifier=None,
    voice_names=None,
):
    TimespanMaker.__init__(
        self,
        output_masks=output_masks,
        padding=padding,
        seed=seed,
        timespan_specifier=timespan_specifier,
    )

    if start_talea is not None:
        if not isinstance(start_talea, rhythmmakertools.Talea):
            start_duration = durationtools.Duration(start_talea)
            counts = [start_duration.numerator]
            denominator = start_duration.denominator
            start_talea = rhythmmakertools.Talea(
                counts=counts,
                denominator=denominator,
            )
        assert isinstance(start_talea, rhythmmakertools.Talea)
        assert start_talea.counts
        assert all(0 < x for x in start_talea.counts)
        self._start_talea = start_talea

    if start_groupings is not None:
        if not isinstance(start_groupings, collections.Sequence):
```
start_groupings = (start_groupings,)
start_groupings = tuple(int(x) for x in start_groupings)
assert len(start_groupings)
assert all(0 < x for x in start_groupings)
self._start_groupings = start_groupings

if stop_talea is not None:
    if not isinstance(stop_talea, rhythmmakertools.Talea):
        stop_duration = durationtools.Duration(stop_talea)
        denominator = stop_duration.denominator
        stop_talea = rhythmmakertools.Talea(
            counts=stop_duration.numerator,
            denominator=denominator,
        )
    assert isinstance(stop_talea, rhythmmakertools.Talea)
    assert stop_talea.counts
    assert all(0 < x for x in stop_talea.counts)
self._stop_talea = stop_talea

if stop_groupings is not None:
    if not isinstance(stop_groupings, collections.Sequence):
        stop_groupings = (stop_groupings,)
        stop_groupings = tuple(int(x) for x in stop_groupings)
    assert len(stop_groupings)
    assert all(0 < x for x in stop_groupings)
self._stop_groupings = stop_groupings

if labels is not None:
    if isinstance(labels, str):
        labels = (labels,)
        labels = tuple(str(_) for _ in labels)
    self._labels = labels

if voice_names is not None:
    voice_names = tuple(voice_names)
self._voice_names = voice_names

### PRIVATE METHODS ###

def _collect_preexisting_timespans(
    self,
    target_timespan=None,
    timespan_inventory=None,
):
    import consort
    preexisting_timespans = timespantools.TimespanInventory()
    for timespan in timespan_inventory:
        if not isinstance(timespan, consort.PerformedTimespan):
            continue
        if (self.voice_names and
            timespan.voice_name not in self.voice_names
        ):
```python
self.labels:
self.labels:
self.labels:
def _make_timespans(
    self,
    layer=None,
    music_specifiers=None,
    target_timespan=None,
    timespan_inventory=None,
    
new_timespans = TimespanInventory()
if not self.voice_names and not self.labels:
    return new_timespans

start_talea = self.start_talea
if start_talea is None:
    start_talea = Talea((0,), 1)
    start_talea = Cursor(start_talea)

start_groupings = self.start_groupings
if start_groupings is None:
    start_groupings = (1,)
    start_groupings = Cursor(start_groupings)

stop_talea = self.stop_talea
if stop_talea is None:
    stop_talea = Talea((0,), 1)
    stop_talea = Cursor(stop_talea)

stop_groupings = self.stop_groupings
if stop_groupings is None:
    stop_groupings = (1,)
    stop_groupings = Cursor(stop_groupings)

if self.seed:
    if self.seed < 0:
        for _ in range(abs(self.seed)):
            start_talea.backtrack()
            start_groupings.backtrack()
            stop_talea.backtrack()
            stop_groupings.backtrack()
    else:
        continue
if not self.labels:
    preexisting_timespans.append(timespan)
elif not hasattr(timespan, 'music_specifier') or 
    not timespan.music_specifier or 
    not timespan.music_specifier.labels:
    continue
elif any(label in timespan.music_specifier.labels
    for label in self.labels):
    preexisting_timespans.append(timespan)
    return preexisting_timespans & target_timespan
```
context_counter = collections.Counter()
preexisting_timespans = self._collect_preexisting_timespans(
    target_timespan=target_timespan,
    timespan_inventory=timespan_inventory,
)
new_timespan_mapping = {}
for group_index, group in enumerate(
    preexisting_timespans.partition(True)
):
    for context_name, music_specifier in music_specifiers.items():
        if context_name not in new_timespan_mapping:
            continue
        new_timespan_mapping[context_name] = group.timespan
    for context_name, music_specifier in music_specifiers.items():
        if context_name not in new_timespan_mapping:
            new_timespan_mapping[context_name] = \
            timespantools.TimespanInventory()
        context_seed = context_counter[context_name]
        start_durations = []
        for _ in range(next(start_groupings)):
            start_durations.append(next(start_talea))
        stop_durations = []
        for _ in range(next(stop_groupings)):
            stop_durations.append(next(stop_talea))
        start_timespans, stop_timespans = (), ()
        if start_durations:
            start_timespans = music_specifier(
                durations=start_durations,
                layer=layer,
                output_masks=self.output_masks,
                padding=self.padding,
                seed=context_seed,
                start_offset=group.start_offset,
                timespan_specifier=self.timespan_specifier,
                voice_name=context_name,
            )
            context_counter[context_name] += 1
        if stop_durations:
            stop_timespans = music_specifier(
                durations=stop_durations,
                layer=layer,
                output_masks=self.output_masks,
                padding=self.padding,
                seed=context_seed,
                start_offset=group.stop_offset,
                timespan_specifier=self.timespan_specifier,
                voice_name=context_name,
            )
            context_counter[context_name] += 1
if start_timespans and stop_timespans:
    start_timespans & group_timespan
    new_timespan_mapping[context_name].extend(start_timespans)
    new_timespan_mapping[context_name].extend(stop_timespans)
    for context_name, timespans in new_timespan_mapping.items():
        timespans.compute_logical_or()
        new_timespans.extend(timespans)
    return new_timespans

### PUBLIC PROPERTIES ###

@property
def labels(self):
    return self._labels

@property
def start_talea(self):
    return self._start_talea

@property
def stop_talea(self):
    return self._stop_talea

@property
def start_groupings(self):
    return self._start_groupings

@property
def stop_groupings(self):
    return self._stop_groupings

@property
def voice_names(self):
    return self._voice_names

A.8 consort.tools.ChordExpression
>>> chord_expression = consort.ChordExpression(
...    arpeggio_direction=Down,
...    chord_expr=(-1, 3, 7),
...)

>>> print(format(chord_expression))
consort.tools.ChordExpression(
    chord_expr=pitchtools.IntervalSegment(
        (pitchtools.NumberedInterval(-1),
         pitchtools.NumberedInterval(3),
         pitchtools.NumberedInterval(7),
        ),
        item_class=pitchtools.NumberedInterval,
    ),
    arpeggio_direction=Down,
)
::

>>> staff = Staff(r"c'4 d'4 \p ~\accent \p -\accent \p ~\accent \p ~\accent \p ~\accent a'4")
>>> pitch_range = pitchtools.PitchRange.from_pitches('C3', 'C6')
>>> attach(pitch_range, staff, scope=Staff)
>>> logical_tie = inspect_(staff[1]).get_logical_tie()
>>> chord_expression(logical_tie)

>>> print(format(staff))
\new Staff {
   c'4 \arpeggioArrowDown
   <cs' f' a'>4 \accent \arpeggio \p -
   <cs' f' a'>4
e'4
}

### CLASS VARIABLES ###

__slots__ =
    '_arpeggio_direction',
    '_chord_expr',

### INITIALIZER ###

def __init__(
    self,
    chord_expr=None,
    arpeggio_direction=None,
):
    assert arpeggio_direction in (Up, Down, Center, None)
    if chord_expr is not None:
        assert len(chord_expr)
        prototype = (pitchtools.IntervalSegment, pitchtools.PitchSegment)
        if isinstance(chord_expr, prototype):
result = chord_expr

elif isinstance(chord_expr, str):
    result = pitchtools.PitchSegment(chord_expr)
else:
    try:
        result = sorted(chord_expr)
    except:
        result = pitchtools.IntervalSegment(result)
    chord_expr = result
    self._arpeggio_direction = arpeggio_direction
    self._chord_expr = chord_expr

### SPECIAL METHODS ###

def __call__(
    self,
    logical_tie,
    pitch_range=None,
):
    assert isinstance(logical_tie, selectiontools.LogicalTie), logical_tie
    if isinstance(self.chord_expr, pitchtools.IntervalSegment):
        pitches = self._get_pitches_from_intervals(
            logical_tie.head.written_pitch,
            pitch_range,
            logical_tie,
            logical_tie,
        )
    else:
        pitches = self.chord_expr
    if len(pitches) == 2:
        interval = pitches[0] - pitches[1]
        if interval.quality_string in ('augmented', 'diminished'):
            chord = scoretools.Chord(pitches, 1)
            mutate(chord).respell_with_sharps()
            pitches = chord.written_pitches
            interval = pitches[0] - pitches[1]
        if interval.quality_string in ('augmented', 'diminished'):
            chord = scoretools.Chord(pitches, 1)
            mutate(chord).respell_with_flats()
            pitches = chord.written_pitches
        for i, leaf in enumerate(logical_tie):
            chord = scoretools.Chord(leaf)
            chord.written_pitches = pitches
            self._replace(leaf, chord)
        if not i and self.arpeggio_direction is not None:
            arpeggio = indicatortools.Arpeggio(self.arpeggio_direction)
            attach(arpeggio, chord)

### PRIVATE METHODS ###

@staticmethod
def _score_pitch_set(pitch_set):
    buckets = {}
    for pitch in pitch_set:
if pitch.diatonic_pitch_number not in buckets:
    buckets[pitch.diatonic_pitch_number] = set()
    buckets[pitch.diatonic_pitch_number].add(pitch)
penalty = 0
for diatonic_pitch_number, bucket in buckets.items():
    penalty = penalty + (len(bucket) - 1)
return penalty

@staticmethod
def _flip_accidental(pitch):
    if not pitch.alteration_in_semitones:
        return pitch
    elif 0 < pitch.alteration_in_semitones:
        return pitch.respell_with_flats()
    return pitch.respell_with_sharps()

@staticmethod
def _reshape_pitch_set(pitch_set):
    altered = {}
    unaltered = {}
    for pitch in pitch_set:
        diatonic_pitch_number = pitch.diatonic_pitch_number
        if pitch.alteration_in_semitones:
            if diatonic_pitch_number not in altered:
                altered[diatonic_pitch_number] = []
                altered[diatonic_pitch_number].append(pitch)
            else:
                unaltered[diatonic_pitch_number] = pitch
        else:
            unaltered[diatonic_pitch_number] = pitch
    new_altered = {}
    for diatonic_pitch_number, altered_pitches in altered.items():
        while altered_pitches:
            altered_pitch = altered_pitches.pop()
            if diatonic_pitch_number not in unaltered and \
                diatonic_pitch_number not in new_altered:
                if diatonic_pitch_number not in new_altered:
                    new_altered[diatonic_pitch_number] = []
                    new_altered[diatonic_pitch_number].append(altered_pitch)
                else:
                    new_pitch = ChordExpression._flip_accidental(altered_pitch)
                    new_diatonic_pitch_number = new_pitch.diatonic_pitch_number
                    if new_diatonic_pitch_number not in new_altered:
                        new_altered[new_diatonic_pitch_number] = []
                        new_altered[new_diatonic_pitch_number].append(new_pitch)
            result = set(unaltered.values())
            for altered_pitches in new_altered.values():
                result.update(altered_pitches)
            result = pitchtools.PitchSet(result)
            penality = penalty + (len(bucket) - 1)
        return result

@staticmethod
def _respell_pitch_set(pitch_set):
    '''Respell pitch set.'
>>> pitch_set = pitchtools.PitchSet("c' e' g'")
>>> consort.ChordExpression._respell_pitch_set(pitch_set)
PitchSet(['c''', 'e''', 'g'''])

::

>>> pitch_set = pitchtools.PitchSet("c' e' g' c'' cs'' g'''")
>>> consort.ChordExpression._respell_pitch_set(pitch_set)
PitchSet(['c''', 'e''', 'g''', 'c'''', 'df''', 'g'''''])

::

>>> pitch_set = pitchtools.PitchSet("bf d' f' fs' bf' f'''")
>>> consort.ChordExpression._respell_pitch_set(pitch_set)
PitchSet(['bf''', 'd''', 'f''', 'gf''', 'bf''', 'f''''])

::

>>> pitch_set = pitchtools.PitchSet("b' c'' cs'' d''")
>>> consort.ChordExpression._respell_pitch_set(pitch_set)
PitchSet(['b''', 'c''', 'cs''', 'd'''])

::

>>> pitch_set = pitchtools.PitchSet("bf f' f' f' f' f'")
>>> consort.ChordExpression._respell_pitch_set(pitch_set)
PitchSet(['b', 'c', 'df'])

::

initial_score = ChordExpression._score_pitch_set(pitch_set)
if not initial_score:
    return pitch_set
flat_pitch_set = pitchtools.PitchSet(
    _._respell_with_flats()
    for _ in pitch_set
)
flat_score = ChordExpression._score_pitch_set(flat_pitch_set)
if not flat_score:
    return flat_pitch_set
sharp_pitch_set = pitchtools.PitchSet(
    _._respell_with_sharps()
    for _ in pitch_set
)
sharp_score = ChordExpression._score_pitch_set(sharp_pitch_set)
if not sharp_score:
    return sharp_pitch_set
scored_pitch_sets = [

463
scored_pitch_sets.sort()
reshaped_pitch_set = ChordExpression._reshape_pitch_set(
    scored_pitch_sets[0][1])
reshaped_score = ChordExpression._score_pitch_set(reshaped_pitch_set)
scored_pitch_sets.append((reshaped_score, reshaped_pitch_set))
scored_pitch_sets.sort()
return scored_pitch_sets[0][1]

def _get_pitches_from_intervals(self, base_pitch, pitch_range, logical_tie):
    import consort
    chord_expr = self.chord_expr or ()
    new_chord_expr = chord_expr
    if pitch_range is not None:
        if base_pitch not in pitch_range:
            print('Voice:', consortSEGMENTMAKER.logical_tie_to_voice(
                logical_tie).name)
            print('Pitch range:', pitch_range)
            print('Base pitch:', base_pitch)
            print('Chord expression:', chord_expr)
            print('MusicSpec:')
            print(format(  # consortSEGMENTMAKER.logical_tie_to_music_specifier(
                logical_tie)))
            raise Exception

        sorted_intervals = sorted(chord_expr, key=lambda x: x.semitones)
        maximum = sorted_intervals[-1]
        maximum_pitch = base_pitch.transpose(maximum)
        minimum = sorted_intervals[0]
        minimum_pitch = base_pitch.transpose(minimum)
        if maximum_pitch not in pitch_range:
            new_chord_expr = [x - maximum for x in chord_expr]
        elif minimum_pitch not in pitch_range:
            new_chord_expr = [x - minimum for x in chord_expr]

    pitches = [base_pitch.transpose(x) for x in new_chord_expr]
pitches = [pitchtools.NamedPitch(float(x)) for x in pitches]
    if pitch_range is not None:
        if not all(pitch in pitch_range for pitch in pitches):
            print('Voice:', consortSEGMENTMAKER.logical_tie_to_voice(
                logical_tie).name)
            print('Pitch range:', pitch_range)
            print('Base pitch:', base_pitch)
            print('Chord expression:', chord_expr)
            print('Resulting pitches:', pitches)
            print('MusicSpec:')
            print(format(  # consortSEGMENTMAKER.logical_tie_to_music_specifier(
                logical_tie)))
            raise Exception
pitch_set = self._respell_pitch_set(pitchtools.PitchSet(pitches))
pitches = pitchtoolsPitchSegment(sorted(pitch_set))

return pitches

### PUBLIC PROPERTIES ###

@property
def arpeggio_direction(self):
    return self._arpeggio_direction

@property
def chord_expr(self):
    return self._chord_expr

A.9  consort.tools.ChordSpecifier

# -*- encoding: utf-8 -*-
from abjad.tools import abctools

class ChordSpecifier(abctools.AbjadValueObject):
    __slots__ = ()

A.10  consort.tools.ClefSpanner

# -*- encoding: utf-8 -*-
from __future__ import print_function
from abjad import inspect_
from abjadtools import indicatortools
from abjadtools import scoretools
from abjadtools import spannertools

class ClefSpanner(spannertools.Spanner):
    r'''Clef spanner.'
>>> clef_spanner = consort.ClefSpanner('percussion')
>>> attach(clef_spanner, staff[2:-2])
>>> print(format(staff))
\new Staff {
  \clef "treble"
  c'4
d'4
  \clef "percussion"
e'4
f'4
g'4
a'4
  \clef "treble"
b'4
c''4
}::

>>> staff = Staff("r4 c'4 d'4 r4 e'4 f'4 r4")
>>> clef = Clef('treble')
>>> attach(clef, staff[0])
>>> clef_spanner = consort.ClefSpanner('percussion')
>>> attach(clef_spanner, staff[1:3])
>>> clef_spanner = consort.ClefSpanner('percussion')
>>> attach(clef_spanner, staff[4:6])
>>> print(format(staff))
\new Staff {
  \clef "treble"
r4
  \clef "percussion"
c'4
d'4
r4
e'4
f'4
g'4
a'4
  \clef "treble"
r4
}::

'''

### CLASS VARIABLES ###

__slots__ = (}
'._clef',
)

### INITIALIZER ###

def __init__(
    self, clef='percussion', overrides=None,):
    spannertools.Spanner.__init__(
        self, overrides=overrides,
    )
clef = indicatortools.Clef(clef)
self._clef = clef

### SPECIAL METHODS ###

def __getnewargs__(self):
    r'''Gets new arguments of spanner.
    Returns empty tuple.
    '''
    return (self.clef,
)

### PRIVATE METHODS ###

def _copy_keyword_args(self, new):
    new._clef = self.clef

def _get_lilypond_format_bundle(self, leaf):
    import consort
    lilypond_format_bundle = self._get_basic_lilypond_format_bundle(leaf)
    prototype = (scoretools.Note, scoretools.Chord, type(None))
    first_leaf = self._get_leaves()[0]
current_clef = inspect_(first_leaf).get_effective(indicatortools.Clef)
set_clef = False
reset_clef = False

if self._is_my_only_leaf(leaf):
    consort.debug('ONLY', leaf)
    if self.clef != current_clef:
        set_clef = True
        reset_clef = True
previous_leaf = inspect_(leaf).get_leaf(-1)
consort.debug('\tP', previous_leaf)
while not isinstance(previous_leaf, prototype):
previous_leaf = inspect_(previous_leaf).get_leaf(-1)
consort.debug('\tP', previous_leaf)

if previous_leaf is not None:
    spanners = inspect_(previous_leaf).get_spanners(type(self))
    spanners = tuple(spanners)
    if spanners:
        consort.debug('\tPREV?', spanners)
        if spanners[0].clef == self.clef:
            set_clef = False

next_leaf = inspect_(leaf).get_leaf(1)
consort.debug('\tN', next_leaf)
while not isinstance(next_leaf, prototype):
    next_leaf = inspect_(next_leaf).get_leaf(1)
    consort.debug('\tN', next_leaf)
    if next_leaf is not None:
        spanners = inspect_(next_leaf).get_spanners(type(self))
        spanners = tuple(spanners)
        if spanners:
            consort.debug('\tNEXT?', spanners)
            if spanners[0].clef == self.clef:
                reset_clef = False

elif self._is_my_first_leaf(leaf):
    consort.debug('FIRST', leaf)
    if self.clef != current_clef:
        set_clef = True

previous_leaf = inspect_(leaf).get_leaf(-1)
consort.debug('\tP', previous_leaf)
while not isinstance(previous_leaf, prototype):
    previous_leaf = inspect_(previous_leaf).get_leaf(-1)
    consort.debug('\tP', previous_leaf)
    if previous_leaf is not None:
        spanners = inspect_(previous_leaf).get_spanners(type(self))
        spanners = tuple(spanners)
        if spanners:
            consort.debug('\tPREV?', spanners)
            if spanners[0].clef == self.clef:
                set_clef = False

elif self._is_my_last_leaf(leaf):
    consort.debug('LAST', leaf)
    if self.clef != current_clef and current_clef is not None:
        reset_clef = True

next_leaf = inspect_(leaf).get_leaf(1)
consort.debug('\tN', next_leaf)
while not isinstance(next_leaf, prototype):
    next_leaf = inspect_(next_leaf).get_leaf(1)
    consort.debug('\tN', next_leaf)
    if next_leaf is not None:
        spanners = inspect_(next_leaf).get_spanners(type(self))
        spanners = tuple(spanners)
if spanners:
    consort.debug('
NEXT?', spanners)
if spanners[0].clef == self.clef:
    reset_clef = False

if set_clef:
    string = format(self.clef, 'lilypond')
    lilypond_format_bundle.before.indicators.append(string)

if reset_clef and current_clef is not None:
    string = format(current_clef, 'lilypond')
    lilypond_format_bundle.after.indicators.append(string)

return lilypond_format_bundle

### PUBLIC PROPERTIES ###

@property
def clef(self):
    return self._clef

A.11 consort.tools.ClefSpannerExpression

```python
# -*- encoding: utf-8 -*-
from __future__ import print_function
from abjad import attach
from abjad import inspect_
from abjad.tools import abctools
from abjad.tools import scoretools

class ClefSpannerExpression(abctools.AbjadValueObject):
    r'''A clef spanner expression.'''

    ### CLASS VARIABLES ###

    __slots__ = ()

    ### INITIALIZER ###

def __init__(self):
    pass

    ### SPECIAL METHODS ###

def __call__(self, music, name=None):
    import consort
    leaves = music.select_leaves()
    weights = []
    weighted_pitches = []
    for leaf in leaves:
        weight = float(inspect_(leaf).get_duration())
        if isinstance(leaf, scoretools.Note):
```
pitch = float(leaf.written_pitch)
weighted_pitch = pitch * weight
weights.append(weight)
weighted_pitches.append(weighted_pitch)
elif isinstance(leaf, scoretools.Chord):
    for pitch in leaf.written_pitches:
        pitch = float(pitch)
        weighted_pitch = pitch * weight
        weighted_pitches.append(weighted_pitch)
        weights.append(weight)
    sum_of_weights = sum(weights)
    sum_of_weighted_pitches = sum(weighted_pitches)
    weighted_average = sum_of_weighted_pitches / sum_of_weights
    if weighted_average < 0:
        clef_spanner = consort.ClefSpanner('bass')
    else:
        clef_spanner = consort.ClefSpanner('treble')
    attach(clef_spanner, music, name=name)

def __init__(self, include_inner_leaves=False, overrides=None):
    spannertools.Spanner.__init__(self,
        include_inner_leaves=False,
        overrides=None,
    )
    spannertools.Spanner.__init__(
        self,
        include_inner_leaves=False,
        overrides=None,
    )
A.12  consort.tools.ComplexPianoPedalSpanner
def _copy_keyword_args(self, new):
    new._include_inner_leaves = self.include_inner_leaves

def _get_lilypond_format_bundle(self, leaf):
    lilypond_format_bundle = self._get_basic_lilypond_format_bundle(leaf)
    if self._is_my_first_leaf(leaf):
        string = r'\sustainOn'
        lilypond_format_bundle.right.spanner_starts.append(string)
    elif self.include_inner_leaves and not self._is_my_last_leaf(leaf):
        string = r'\sustainOff \sustainOn'
        lilypond_format_bundle.right.spanner_starts.append(string)
    if self._is_my_last_leaf(leaf):
        string = r'<> \sustainOff'
        lilypond_format_bundle.after.indicators.append(string)
    return lilypond_format_bundle

A.13 consort.tools.ComplexTextSpanner
... )
>>> attach(spanner_one, staff[:2])
>>> attach(spanner_two, staff[3:])

::

>>> print(format(staff))
\new Staff {
 \once \override TextSpanner.bound-details.left-broken.text = ##f
 \once \override TextSpanner.bound-details.left.text = \markup { foo }
 \once \override TextSpanner.bound-details.right-broken.text = ##f
 \once \override TextSpanner.bound-details.right.text = \markup {
 \draw-line
     #'(0 . -1)
 }
 \once \override TextSpanner.dash-fraction = 1
 \once \override TextSpanner.direction = #up
c'4 \startTextSpan
d'4
<> \stopTextSpan
r4
e'4 _ \markup { bar }
}
::

c\text{container:: example}

::

>>> import consort
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> spanner_one = consort.ComplexTextSpanner(
     ... direction=Up,
     ... markup='foo',
     ... )
>>> spanner_two = consort.ComplexTextSpanner(
     ... direction=Down,
     ... markup='bar',
     ... )
>>> attach(spanner_one, staff[:2])
>>> attach(spanner_two, staff[3:])

::

>>> print(format(staff))
\new Staff {
 \once \override TextSpanner.bound-details.left-broken.text = ##f
 \once \override TextSpanner.bound-details.left.text = \markup { foo }
 \once \override TextSpanner.bound-details.right-broken.text = ##f
 \once \override TextSpanner.bound-details.right.text = \markup {
 \draw-line
     #'(0 . -1)
 }
 \once \override TextSpanner.dash-fraction = 1
 \once \override TextSpanner.direction = #up
.. container:: example

>>> staff = Staff("c'8 d' e' r r a' b' c''")
>>> spanner_one = consort.ComplexTextSpanner(
...    direction=Up,
...    markup='foo',
...  )
>>> spanner_two = consort.ComplexTextSpanner(
...    direction=Up,
...    markup='foo',
...  )
>>> attach(spanner_one, staff[:3])
>>> attach(spanner_two, staff[5:])

::

>>> print(format(staff))
\new Staff {
  \once \override TextSpanner.bound-details.left-broken.text = ##f
  \once \override TextSpanner.bound-details.left.text = \markup { foo }
  \once \override TextSpanner.bound-details.right-broken.text = ##f
  \once \override TextSpanner.bound-details.right.text = \markup {
    \draw-line
    \override \mark\draw-line\mark\draw-line\mark\draw-line
    #'(0 . -1)
  }
  \once \override TextSpanner.dash-fraction = 1
  \once \override TextSpanner.direction = #up
  c'8 \startTextSpan
  d'8
  e'8
  r8
  r8
  a'8
  b'8
  c''8
  <> \stopTextSpan
}

.. container:: example

>>> staff = Staff("c'8 d' e' f' g' a' b' c''")
>>> spanner_one = consort.ComplexTextSpanner(
...    direction=Up,
>>> spanner_two = consort.ComplexTextSpanner(
...    direction=Down,
...    markup='bar',
... )

>>> spanner_three = consort.ComplexTextSpanner(
...    direction=Up,
...    markup='foo',
... )

>>> attach(spanner_one, staff[:3])
>>> attach(spanner_two, staff[3:5])
>>> attach(spanner_three, staff[5:])

::

>>> print(format(staff))
\new Staff {
\once \override TextSpanner.bound-details.left-broken.text = ##f
\once \override TextSpanner.bound-details.left.text = \markup { foo }
\once \override TextSpanner.bound-details.right-broken.text = ##f
\once \override TextSpanner.bound-details.right.text = \markup {
  \draw-line
    #'(0 . -1)
}
\once \override TextSpanner.dash-fraction = 1
\once \override TextSpanner.direction = #up
c'8 \startTextSpan
d'8
e'8
<> \stopTextSpan
\once \override TextSpanner.bound-details.left-broken.text = ##f
\once \override TextSpanner.bound-details.left.text = \markup { bar }
\once \override TextSpanner.bound-details.right-broken.text = ##f
\once \override TextSpanner.bound-details.right.text = \markup {
  \draw-line
    #'(0 . -1)
}
\once \override TextSpanner.dash-fraction = 1
\once \override TextSpanner.direction = #down
f'8 \startTextSpan
g'8
<> \stopTextSpan
\once \override TextSpanner.bound-details.left-broken.text = ##f
\once \override TextSpanner.bound-details.left.text = \markup { foo }
\once \override TextSpanner.bound-details.right-broken.text = ##f
\once \override TextSpanner.bound-details.right.text = \markup {
  \draw-line
    #'(0 . -1)
}
\once \override TextSpanner.dash-fraction = 1
\once \override TextSpanner.direction = #up
a'8 \startTextSpan
b'8
### CLASS VARIABLES ###

```python
__slots__ = ('_direction', '_markup',)
```

### INITIALIZER ###

```python
def __init__(self, direction=None, markup=None, overrides=None):
    spannertools.Spanner.__init__(self, overrides=overrides,)
    assert direction in (Up, Down, None)
    self._direction = direction
    self._markup = markuptools.Markup(markup)
```

### PRIVATE METHODS ###

```python
def _copy_keyword_args(self, new):
    new._markup = self.markup

def _get_lilypond_format_bundle(self, leaf):
    lilypond_format_bundle = self._get_basic_lilypond_format_bundle(leaf)

    if self._is_my_only_leaf(leaf):
        previous_is_similar = self._previous_spanner_is_similar(leaf)
        next_is_similar = self._next_spanner_is_similar(leaf)

        if previous_is_similar and next_is_similar:
            pass

        elif previous_is_similar:
            self._make_spanner_stop(lilypond_format_bundle)

        elif next_is_similar:
            self._make_spanner_start(lilypond_format_bundle)

        elif leaf.written_duration <= durationtools.Duration(1, 4):
            self._make_markup(lilypond_format_bundle)

        else:
```
elif self._is_my_first_leaf(leaf):
    if not self._previous_spanner_is_similar(leaf):
        self._make_spanner_start(lilypond_format_bundle)

eelif self._is_my_last_leaf(leaf):
    if not self._next_spanner_is_similar(leaf):
        self._make_spanner_stop(lilypond_format_bundle)

return lilypond_format_bundle

def _make_markup(self, lilypond_format_bundle):
    direction = self.direction or Up
    markup = markuptools.Markup(
        self.markup.contents,
        direction,
    )
    lilypond_format_bundle.right.markup.append(markup)

def _make_spanner_start(self, lilypond_format_bundle):
    override = lilypondnametools.LilyPondGrobOverride(
        grob_name='TextSpanner',
        is_once=True,
        property_path=('[bound-details]', 'left', 'text'),
        value=self.markup,
    )
    lilypond_format_bundle.update(override)
    override = lilypondnametools.LilyPondGrobOverride(
        grob_name='TextSpanner',
        is_once=True,
        property_path=('[bound-details]', 'left-broken', 'text'),
        value=False,
    )
    lilypond_format_bundle.update(override)
    override = lilypondnametools.LilyPondGrobOverride(
        grob_name='TextSpanner',
        is_once=True,
        property_path=('[bound-details]', 'right', 'text'),
        value=markuptools.Markup(r'\draw-line #\'(0 . -1)"
    )
    lilypond_format_bundle.update(override)
    override = lilypondnametools.LilyPondGrobOverride(
        grob_name='TextSpanner',
        is_once=True,
        property_path=('[bound-details]', 'right-broken', 'text'),
        value=False,
    )
    lilypond_format_bundle.update(override)
    override = lilypondnametools.LilyPondGrobOverride(
        grob_name='TextSpanner',
        is_once=True,
        property_path=('[dash-fraction]'),
    )

241    self._make_spanner_start(lilypond_format_bundle)
242    self._make_spanner_stop(lilypond_format_bundle)
243
244    if self._is_my_first_leaf(leaf):
245        if not self._previous_spanner_is_similar(leaf):
246            self._make_spanner_start(lilypond_format_bundle)
247
248    if self._is_my_last_leaf(leaf):
249        if not self._next_spanner_is_similar(leaf):
250            self._make_spanner_stop(lilypond_format_bundle)
251
252    return lilypond_format_bundle
253
254    def _make_markup(self, lilypond_format_bundle):
255        direction = self.direction or Up
256        markup = markuptools.Markup(
257            self.markup.contents,
258            direction,
259        )
260        lilypond_format_bundle.right.markup.append(markup)
261
262    def _make_spanner_start(self, lilypond_format_bundle):
263        override = lilypondnametools.LilyPondGrobOverride(
264            grob_name='TextSpanner',
265            is_once=True,
266            property_path=('[bound-details]', 'left', 'text'),
267            value=self.markup,
268        )
269        lilypond_format_bundle.update(override)
270        override = lilypondnametools.LilyPondGrobOverride(
271            grob_name='TextSpanner',
272            is_once=True,
273            property_path=('[bound-details]', 'left-broken', 'text'),
274            value=False,
275        )
276        lilypond_format_bundle.update(override)
277        override = lilypondnametools.LilyPondGrobOverride(
278            grob_name='TextSpanner',
279            is_once=True,
280            property_path=('[bound-details]', 'right', 'text'),
281            value=markuptools.Markup(r'\draw-line #\'(0 . -1)"
282        )
283        lilypond_format_bundle.update(override)
284        override = lilypondnametools.LilyPondGrobOverride(
285            grob_name='TextSpanner',
286            is_once=True,
287            property_path=('[bound-details]', 'right-broken', 'text'),
288            value=False,
289        )
290        lilypond_format_bundle.update(override)
291        override = lilypondnametools.LilyPondGrobOverride(
292            grob_name='TextSpanner',
293            is_once=True,
294            property_path=('[dash-fraction]'),,
value=1,

lilypond_format_bundle.update(override)
if self.direction is not None:
    override = lilypondnametools.LilyPondGrobOverride(
        grob_name='TextSpanner',
        is_once=True,
        property_path=('direction',),
        value=self.direction,
    )
    lilypond_format_bundle.update(override)
string = r'\startTextSpan'
lilypond_format_bundle.right.spanner_starts.append(string)
def _make_spanner_stop(self, lilypond_format_bundle):
    string = r'<> \stopTextSpan'
lilypond_format_bundle.after.indicators.append(string)

# def _next_spanner_is_similar(self, leaf):
#     next_leaf = leaf._get_leaf(1)
#     next_spanner = None
#     next_spanner_is_similar = False
#     if next_leaf is not None:
#         spanners = next_leaf._get_spanners(type(self))
#         if spanners:
#             assert len(spanners) == 1
#             next_spanner = tuple(spanners)[0]
#             if next_spanner.direction == self.direction:
#                 if next_spanner.markup == self.markup:
#                     next_spanner_is_similar = True
#     return next_spanner_is_similar

def _next_spanner_is_similar(self, leaf):
    leaf_prototype = (scoretools.Note, scoretools.Chord)
    next_spanner = None
    next_spanner_is_similar = False
    for index in range(1, 5):
        next_leaf = leaf._get_leaf(index)
        if next_leaf is None:
            break
        has_spanner = next_leaf._has_spanner(type(self))
        if not has_spanner:
            if isinstance(next_leaf, leaf_prototype):
                break
        continue
    next_spanner = next_leaf._get_spanner(type(self))
    if next_spanner.direction != self.direction:
        break
    if next_spanner.markup != self.markup:
        break
    next_spanner_is_similar = True
    return next_spanner_is_similar

def _previous_spanner_is_similar(self, leaf):
leaf_prototype = (scoretools.Note, scoretools.Chord)
previous_spanner = None
previous_spanner_is_similar = False
for index in range(1, 5):
    previous_leaf = leaf._get_leaf(-index)
    if previous_leaf is None:
        break
    has_spanner = previous_leaf._has_spanner(type(self))
    if not has_spanner:
        if isinstance(previous_leaf, leaf_prototype):
            break
        continue
    previous_spanner = previous_leaf._get_spanner(type(self))
    if previous_spanner.direction != self.direction:
        break
    if previous_spanner.markup != self.markup:
        break
    previous_spanner_is_similar = True
return previous_spanner_is_similar

# def _previous_spanner_is_similar(self, leaf):
#     previous_leaf = leaf._get_leaf(-1)
#     previous_spanner = None
#     previous_spanner_is_similar = False
#     if previous_leaf is not None:
#         spanners = previous_leaf._get_spanners(type(self))
#         if spanners:
#             assert len(spanners) == 1
#             previous_spanner = tuple(spanners)[0]
#             if previous_spanner.direction == self.direction:
#                 if previous_spanner.markup == self.markup:
#                     previous_spanner_is_similar = True
#     return previous_spanner_is_similar

### PUBLIC PROPERTIES ###

@property
def direction(self):
    return self._direction

@property
def markup(self):
    return self._markup

A.14  consort.tools.CompositeMusicSpecifier
class CompositeMusicSpecifier(HashCachingObject):
    r'''A composite music specifier.

>>> import consort
>>> music_specifier = consort.CompositeMusicSpecifier(
    ... primary_musicSpecifier='one',
    ... primary_voice_name='Viola 1 RH',
    ... rotation_indices=(0, 1, -1),
    ... secondary_voice_name='Viola 1 LH',
    ... secondary_music_specifier=consort.MusicSpecifierSequence(
        ... application_rate='phrase',
        ... music_specifiers=['two', 'three', 'four'],
        ... ),
    ... )

>>> print(format(music_specifier))

consort.tools.CompositeMusicSpecifier(
    primary_music_specifier=consort.tools.MusicSpecifierSequence(  
        music_specifiers=datastructuretools.CyclicTuple([  
            'one'
        ]),
    ),
    primary_voice_name='Viola 1 RH',
    rotation_indices=(0, 1, -1),
    secondary_music_specifier=consort.tools.MusicSpecifierSequence(  
        application_rate='phrase',
        music_specifiers=datastructuretools.CyclicTuple([  
            'two', 'three', 'four'
        ]),
    ),
    secondary_voice_name='Viola 1 LH',
)

...:

>>> durations = [1, 2]
>>> timespans = music_specifier(
    ... durations=durations,
    ... layer=1,
    ... )

>>> print(format(timespans))
timespantools.TimespanInventory(
    [  
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(1, 1),
            layer=1,
            music_specifier='two',
            voice_name='Viola 1 LH',
        ),
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(1, 1),
        ),
    ]
)
layer=1,
music_specifier='one',
voice_name='Viola 1 RH',
),
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(1, 1),
    stop_offset=durationtools.Offset(3, 1),
    layer=1,
    music_specifier='two',
    voice_name='Viola 1 LH',
),
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(1, 1),
    stop_offset=durationtools.Offset(3, 1),
    layer=1,
    music_specifier='one',
    voice_name='Viola 1 RH',
),
]
::
::
>>> durations = [1, 2]
>>> timespans = music_specifier(
    ... durations=durations,
    ... layer=2,
    ... seed=1,
    ... )
>>> print(format(timespans))
timespantools.TimespanInventory([
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(1, 1),
        layer=2,
        music_specifier='one',
        voice_name='Viola 1 RH',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(2, 1),
        layer=2,
        music_specifier='three',
        voice_name='Viola 1 LH',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(1, 1),
        stop_offset=durationtools.Offset(3, 1),
        layer=2,
        music_specifier='one',
        voice_name='Viola 1 RH',
    ),
    consort.tools.PerformedTimespan(}
start_offset=durationtools.Offset(2, 1),
stop_offset=durationtools.Offset(3, 1),
layer=2,
musicSpecifier='three',
voice_name='Viola 1 LH',
],
)

>>> durations = [1, 2]
>>> timespans = musicSpecifier(
...    durations=durations,
...    layer=3,
...    padding=1,
...    seed=2,
...)

>>> print(format(timespans))
timespantools.TimespanInventory(
    [
        consort.tools.SilentTimespan(
            start_offset=durationtools.Offset(-1, 1),
            stop_offset=durationtools.Offset(0, 1),
            layer=3,
            voice_name='Viola 1 RH',
        ),
        consort.tools.SilentTimespan(
            start_offset=durationtools.Offset(-1, 1),
            stop_offset=durationtools.Offset(0, 1),
            layer=3,
            voice_name='Viola 1 LH',
        ),
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(1, 1),
            layer=3,
            musicSpecifier='one',
            voice_name='Viola 1 RH',
        ),
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(2, 1),
            layer=3,
            musicSpecifier='four',
            voice_name='Viola 1 LH',
        ),
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(1, 1),
            stop_offset=durationtools.Offset(3, 1),
            layer=3,
            musicSpecifier='one',
            voice_name='Viola 1 RH',
        ),
```python
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(2, 1),
    stop_offset=durationtools.Offset(3, 1),
    layer=3,
    music_specifier='four',
    voice_name='Viola 1 LH',
),
consort.tools.SilentTimespan(
    start_offset=durationtools.Offset(3, 1),
    stop_offset=durationtools.Offset(4, 1),
    layer=3,
    voice_name='Viola 1 RH',
),
consort.tools.SilentTimespan(
    start_offset=durationtools.Offset(3, 1),
    stop_offset=durationtools.Offset(4, 1),
    layer=3,
    voice_name='Viola 1 LH',
)
]

### CLASS VARIABLES ###

__slots__ =
    ('_discard_inner_offsets',
     '_primary_music_specifier',
     '_primary_voice_name',
     '_rotation_indices',
     '_secondary_music_specifier',
     '_secondary_voice_name',
)

### INITIALIZER ###

def __init__(
    self,
    discard_inner_offsets=None,
    primary_music_specifier=None,
    primary_voice_name=None,
    rotation_indices=None,
    secondary_music_specifier=None,
    secondary_voice_name=None,
):
    import consort
    HashCachingObject.__init__(self)
    prototype = consort.MusicSpecifierSequence
    if discard_inner_offsets is not None:
        discard_inner_offsets = bool(discard_inner_offsets)
        self._discard_inner_offsets = discard_inner_offsets
    if not isinstance(primary_music_specifier, prototype):
        primary_music_specifier = consort.MusicSpecifierSequence(
```

music_specifiers=primary_music_specifier,
)
self._primary_music_specifier = primary_music_specifier
if primary_voice_name is not None:
    primary_voice_name = str(primary_voice_name)
    self._primary_voice_name = primary_voice_name
if rotation_indices is not None:
    if not isinstance(rotation_indices, collections.Sequence):
        rotation_indices = int(rotation_indices)
        rotation_indices = (rotation_indices,)
    rotation_indices = tuple(rotation_indices)
    self._rotation_indices = rotation_indices
if not isinstance(secondary_music_specifier, prototype):
    secondary_music_specifier = consort.MusicSpecifierSequence(
        music_specifiers=secondary_music_specifier,
    )
    self._secondary_music_specifier = secondary_music_specifier
if secondary_voice_name is not None:
    secondary_voice_name = str(secondary_voice_name)
    self._secondary_voice_name = secondary_voice_name

### PUBLIC METHODS ###

def __call__(
    self,
    durations=None,
    layer=None,
    output_masks=None,
    padding=None,
    seed=None,
    start_offset=None,
    timespan_specifier=None,
    voice_name=None,
):
    seed = seed or 0
    rotation_indices = self.rotation_indices or (0,)
    rotation_indices = datastructuretools.CyclicTuple(rotation_indices)
    primary_durations = durations
    start_offset = start_offset or 0
    if self.discard_inner_offsets:
        secondary_durations = [sum(primary_durations)]
    else:
        secondary_durations = sequencetools.rotate_sequence(
            primary_durations,
            rotation_indices[seed],
        )
    primary_timespans = self.primary_music_specifier(
        durations=primary_durations,
        layer=layer,
        output_masks=output_masks,
        padding=padding,
        seed=seed,
        start_offset=start_offset,
        timespan_specifier=timespan_specifier,
voice_name=self.primary_voice_name,
)

secondary_timespans = self.secondary_music_specifier(
durations=secondary_durations,
layer=layer,
output_masks=output_masks,
padding=padding,
seed=seed,
start_offset=start_offset,
timespanSpecifier=timespanSpecifier,
voice_name=self.secondary_voice_name,
)

timespans = primary_timespans[:] + secondary_timespans[:]
timespans = timespantools.TimespanInventory(timespans)
timespans.sort()
return timespans

### PUBLIC PROPERTIES ###

@property
def discard_inner_offsets(self):
    return self._discard_inner_offsets

@property
def primary_music_specifier(self):
    return self._primary_music_specifier

@property
def primary_voice_name(self):
    return self._primary_voice_name

@property
def rotation_indices(self):
    return self._rotation_indices

@property
def secondary_music_specifier(self):
    return self._secondary_music_specifier

@property
def secondary_voice_name(self):
    return self._secondary_voice_name

A.15  consort.tools.CompositeRhythmMaker

# -*- encoding: utf-8 -*-
from abjad import new
from abjad.tools import abctools
from abjad.tools import durationtools
from abjad.tools import rhythmmakertools

class CompositeRhythmMaker(abctools.AbjadValueObject):
    r'''A composite rhythm-maker.'
>>> import consort

>>> composite_rhythm_maker = consort.CompositeRhythmMaker(
...     default=rhythmmakertools.EvenDivisionRhythmMaker(),
...     first=rhythmmakertools.NoteRhythmMaker(),
...     last=rhythmmakertools.IncisedRhythmMaker(
...         inciseSpecifier=rhythmmakertools.InciseSpecifier(
...             prefix_counts=[0],
...             suffix_talea=[1],
...             suffix_counts=[1],
...             talea_denominator=16,
...         ),
...     ),
...     only=rhythmmakertools.EvenDivisionRhythmMaker(
...         denominators=[32],
...     ),
... )

>>> print(format(composite_rhythm_maker))
consort.tools.CompositeRhythmMaker(
    default=rhythmmakertools.EvenDivisionRhythmMaker(
        denominators=(8,),
        preferred_denominator='from_counts',
    ),
    first=rhythmmakertools.NoteRhythmMaker(),
    last=rhythmmakertools.IncisedRhythmMaker(
        inciseSpecifier=rhythmmakertools.InciseSpecifier(
            prefix_counts=(0,),
            suffix_talea=(1,),
            suffix_counts=(1,),
            talea_denominator=16,
        ),
    ),
    only=rhythmmakertools.EvenDivisionRhythmMaker(
        denominators=(32,),
        preferred_denominator='from_counts',
    ),
)

.. container:: example

    ::

    >>> divisions = [(1, 4), (1, 4), (1, 4), (1, 4)]
    >>> result = composite_rhythm_maker(divisions)
    >>> staff = Staff()
    >>> for x in result:
    ...     staff.extend(x)
    ...    
    >>> print(format(staff))
    \new Staff { 
    c'4 
    ...}
>>> divisions = [(1, 4), (1, 4)]
>>> result = composite_rhythm_maker(divisions)
>>> staff = Staff()
>>> for x in result:
...    staff.extend(x)
...    print(format(staff))
\new Staff {
    \c'4
    \c'8. [
    \c'16 ]
}

>>> divisions = [(1, 4)]
>>> result = composite_rhythm_maker(divisions)
>>> staff = Staff()
>>> for x in result:
...    staff.extend(x)
...    print(format(staff))
\new Staff {
    \c'32 [
    \c'32
    \c'32
    \c'32
    \c'32
    \c'32
    \c'32 ]
}

### CLASS VARIABLES ###

__slots__ = ('_default', '_first', '_last', '_only',)

### INITIALIZER ###

def __init__(self, default=None, first=None, last=None, only=None):
    
    if first is not None:
        assert isinstance(first, rhythmmakertools.RhythmMaker)
    if last is not None:
        assert isinstance(last, rhythmmakertools.RhythmMaker)
    if only is not None:
        assert isinstance(only, rhythmmakertools.RhythmMaker)
    if default is None:
        default = rhythmmakertools.NoteRhythmMaker()
        assert isinstance(default, rhythmmakertools.RhythmMaker)

    self._first = first
    self._last = last
    self._only = only
    self._default = default

### SPECIAL METHODS ###

def __call__(self, divisions, rotation=None):
    divisions = [durationtools.Division(x) for x in divisions]
    result = []
    if not divisions:
        pass
    elif len(divisions) == 1:
        if self.only:
            result.extend(self.only(divisions, rotation=rotation))
    elif self.last:
        result.extend(self.last(divisions, rotation=rotation))
    elif self.first:
        result.extend(self.first(divisions, rotation=rotation))
    else:
        result.extend(self.default(divisions, rotation=rotation))
    elif len(divisions) == 2:
        if self.first and self.last:
            first = self.first(divisions=[divisions[0]], rotation=rotation)
            last = self.last(divisions=[divisions[1]], rotation=rotation)
            result.extend(first)
            result.extend(last)
elif self.first:
    first = self.first(divisions=[divisions[0]], rotation=rotation)
    default = self.default(divisions=[divisions[1]], rotation=rotation)
    result.extend(first)
    result.extend(default)

eif self.last:
    default = self.default(divisions=[divisions[0]], rotation=rotation)
    last = self.last(divisions=[divisions[1]], rotation=rotation)
    result.extend(default)
    result.extend(last)
else:
    default = self.default(divisions=divisions, rotation=rotation)
    result.extend(default)

else:
    if self.first and self.last:
        first = self.first(divisions=[divisions[0]], rotation=rotation)
        default = self.default(divisions=[divisions[1:-1]], rotation=rotation)
        last = self.last(divisions=[divisions[-1]], rotation=rotation)
        result.extend(first)
        result.extend(default)
        result.extend(last)
    elif self.first:
        first = self.first(divisions=[divisions[0]], rotation=rotation)
        default = self.default(divisions=[divisions[1]], rotation=rotation)
        result.extend(first)
        result.extend(default)
    elif self.last:
        default = self.default(divisions=[divisions[0]], rotation=rotation)
        last = self.last(divisions=[divisions[-1]], rotation=rotation)
        result.extend(default)
        result.extend(last)
    else:
        default = self.default(divisions=divisions, rotation=rotation)
        result.extend(default)

return result

def __illustrate__(self, divisions=None):
    r'''Illustrates composite rhythm-maker.

    Returns LilyPond file.

    '''

divisions = divisions or [
    (3, 8),
    (4, 8),
    (3, 16),
    (4, 16),
    (5, 8),
    (2, 4),
    (5, 16),
    (2, 8),
    (7, 8),
]
selections = self(divisions)
lilypond_file = rhythmmakertools.make_lilypond_file(
selections,
    divisions,
)
    return lilypond_file

### PUBLIC METHODS ###

def new(
    self,
    first=None,
    last=None,
    only=None,
    default=None,
    **kwargs
):
    first = first or self.first
    last = last or self.last
    only = only or self.only
    default = default or self.default
    if first is not None:
        first = new(first, **kwargs)
    if last is not None:
        last = new(last, **kwargs)
    if only is not None:
        only = new(only, **kwargs)
    if default is not None:
        default = new(default, **kwargs)
    result = new(
        self,
        first=first,
        last=last,
        only=only,
        default=default,
    )
    return result

### PUBLIC PROPERTIES ###

@property
def first(self):
    return self._first

@property
def last(self):
    return self._last

@property
def only(self):
    return self._only

@property
def default(self):
    return self._default
class ConsortTrillSpanner(spannertools.Spanner):
    r''''A complex trill spanner."

.. container:: example

    >>> staff = Staff("c'4 ~ c'8 d'8 r8 e'8 ~ e'8 r8")
    >>> show(staff) # doctest: +SKIP

.. doctest::

    >>> print(format(staff))
    "new Staff {
        \pitchedTrill
        c'4 ~ \startTrillSpan f'
        c'8 
        \stopTrillSpan
        \pitchedTrill
        d'8 \startTrillSpan g'
        \stopTrillSpan
        r8
        \pitchedTrill"

.. doctest::

    >>> import consort
    >>> complex_trill = consort.ConsortTrillSpanner(
    ...     interval='P4',
    ...     )
    >>> attach(complex_trill, staff.select_leaves())
    >>> show(staff) # doctest: +SKIP

.. doctest::

    >>> print(format(staff))
    "new Staff {
        \pitchedTrill
        c'4 ~ \startTrillSpan f'
        c'8 
        \stopTrillSpan
        \pitchedTrill
        d'8 \startTrillSpan g'
        \stopTrillSpan
        r8
        \pitchedTrill"
```python
\begin{verbatim}
e'8 \startTrillSpan a'
e'8
 <> \stopTrillSpan
 r8
 }

Allows for specifying a trill pitch via a named interval.

Avoids silences.

Restarts the trill on every new pitched logical tie.

'''

### CLASS VARIABLES ###

__slots__ = ('_interval',)

### INITIALIZER ###

def __init__(self, overrides=None, interval=None):
    spannertools.Spanner.__init__(self, overrides=overrides,)
    if interval is not None:
        interval = pitchtools.NamedInterval(interval)
    self._interval = interval

### PRIVATE METHODS ###

def _copy_keyword_args(self, new):
    new._interval = self.interval

def _get_lilypond_format_bundle(self, leaf):
    from abjad.tools import scoretools
    lilypond_format_bundle = self._get_basic_lilypond_format_bundle(leaf)
    prototype = (scoretools.Rest,
                 scoretools.MultimeasureRest,
                 scoretools.Skip,
                 )
    if isinstance(leaf, prototype):
        return lilypond_format_bundle
    logical_tie = inspect_(leaf).get_logical_tie()
    starts_spanner, stops_spanner = False, False
    if leaf.is_logical_tie.head:
        starts_spanner = True
```
after_graces = inspect_(leaf).get_grace_containers('after')
if leaf is logical_tie.tail and not len(after_graces):
    stops_spanner = True
elif self._is_my_last_leaf(leaf):
    stops_spanner = True
if starts_spanner:
    previous_leaf = leaf._get_leaf(-1)
    if previous_leaf is not None:
        after_graces = inspect_(previous_leaf).get_grace_containers('after')
        if after_graces:
            grob_override = lilypondnametools.LilyPondGrobOverride(
                grob_name='TrillSpanner',
                is_once=True,
                property_path=(
                    'bound-details',
                    'left',
                    'padding',
                ),
                value=2,
            )
            string = '\n'.join(grob_override._override_format_pieces)
            lilypond_format_bundle.grob_overrides.append(string)
            if self.interval is not None:
                string = r'\pitchedTrill'
                lilypond_format_bundle.opening.spanners.append(string)
                if hasattr(leaf, 'written_pitch'):
                    written_pitch = leaf.written_pitch
                elif hasattr(leaf, 'written_pitches'):
                    if 0 < self.interval.semitones:
                        written_pitch = max(leaf.written_pitches)
                    elif self.interval.semitones < 0:
                        written_pitch = min(leaf.written_pitches)
                trill_pitch = written_pitch.transpose(self.interval)
                string = r'\startTrillSpan {!s}'.format(trill_pitch)
            else:
                string = r'\startTrillSpan'
                lilypond_format_bundle.right.trill_pitches.append(string)
        if stops_spanner:
            next_leaf = leaf._get_leaf(1)
            if next_leaf is not None:
                string = r'<> \stopTrillSpan'
                lilypond_format_bundle.after.commands.append(string)
            else:
                string = r'\stopTrillSpan'
                lilypond_format_bundle.right.spanner_stops.append(string)
    return lilypond_format_bundle

### PUBLIC PROPERTIES ###
@property
def interval(self):
    r'''Gets optional interval of trill spanner.

    .. container:: example

        >>> import consort
        >>> staff = Staff("c'4 d'4 e'4 f'4")
        >>> interval = pitchtools.NamedInterval('m3')
        >>> complex_trill = consort.ConsortTrillSpanner(
        ...     interval=interval)
        >>> attach(complex_trill, staff[1:-1])
        >>> show(staff) # doctest: +SKIP

    .. doctest::

        >>> print(format(staff))
        \new Staff {
        c'4 pitchedTrill d'4 \startTrillSpan f'
        <> \stopTrillSpan pitchedTrill e'4 \startTrillSpan g'
        <> \stopTrillSpan f'4
        }

    >>> complex_trill.interval
    NamedInterval('+m3')

    return self._interval

A.17  consort.tools.Cursor

# -*- encoding: utf-8 -*-
from abjad.tools import abctools
from abjad.tools import datastructuretools

class Cursor(abctools.AbjadValueObject):
    r'''A cursor.

    .. container:: example

        >>> import consort
        >>> cursor = consort.Cursor([1, 2, 3])
        >>> next(cursor)
16    
17    ::
18    
19    >>> next(cursor)
20    2
21    ::
22    
23    >>> next(cursor)
24    3
25    ::
26    
27    >>> next(cursor)
28    1
29    ::
30    
31    >>> next(cursor)
32    2
33    ::
34    
35    >>> cursor.backtrack()
36    2
37    ::
38    
39    >>> cursor.backtrack()
40    1
41    ::
42    
43    >>> cursor.backtrack()
44    3
45    ::
46    
47    >>> next(cursor)
48    3
49    ::
50    
51    >>> next(cursor)
52    1
53    ::
54    
55    >>> next(cursor)
56    3
57    ::
58    
59    >>> next(cursor)
60    1
61    
62    .. container:: example
63    
64    ::
65    
66    >>> talea = rhythmmakertools.Talea(
67    ...    counts=(2, 1, 3, 2, 4, 1, 1),
68    ...    denominator=16,
>>> cursor = consort.Cursor(talea)
>>> for _ in range(10):
...     next(cursor)
...
Duration(1, 8)
Duration(1, 16)
Duration(3, 16)
Duration(1, 8)
Duration(1, 4)
Duration(1, 16)
Duration(1, 16)
Duration(1, 8)
Duration(1, 16)
Duration(3, 16)

### CLASS VARIABLES ###
__slots__ = ('_sequence', '_index')

### INITIALIZER ###
def __init__(self, sequence=(1, 2, 3), index=None):
    self._sequence = datastructuretools.CyclicTuple(sequence)
    if index is not None:
        index = int(index)
        self._index = index

### SPECIAL METHODS ###
def __iter__(self):
    while True:
        yield self.next()

def __next__(self):
    return self.next()

### PUBLIC METHODS ###
def backtrack(self):
    if not self._sequence:
        return
    if self._index is None:
        self._index = 0
        self._index -= 1
    index = self._index
    return self._sequence[index]

def next(self):

if not self._sequence:
    return
if self._index is None:
    self._index = 1
    return self._sequence[0]
index = self._index
self._index += 1
return self._sequence[index]

### PUBLIC PROPERTIES ###

@property
def index(self):
    return self._index
@property
def sequence(self):
    return self._sequence

A.18 consort.tools.DependentTimespanMaker

# -*- encoding: utf-8 -*-
import collections
from abjad import inspect_
from abjad.tools import datastructuretools
from abjad.tools import durationtools
from abjad.tools import mathtools
from abjad.tools import sequencetools
from abjad.tools import timespantools
from consort.tools.TimespanMaker import TimespanMaker

class DependentTimespanMaker(TimespanMaker):
    r'''A dependent timespan-maker.''

    >>> import consort
    >>> timespan_maker = consort.DependentTimespanMaker(
    ...     include_inner_starts=True,
    ...     include_inner_stops=True,
    ...     voice_names=('Viola Voice',),
    ...     )
    >>> print(format(timespan_maker))
    consort.tools.DependentTimespanMaker(
        include_inner_starts=True,
        include_inner_stops=True,
        voice_names=('Viola Voice',),
        )
>>> timespan_inventory = timespantools.TimespanInventory([
...     consort.tools.PerformedTimespan(
...         voice_name='Viola Voice',
...         start_offset=(1, 4),
...         stop_offset=(1, 1),
...     ),
...     consort.tools.PerformedTimespan(
...         voice_name='Viola Voice',
...         start_offset=(3, 4),
...         stop_offset=(3, 2),
...     ),
... ])

>>> music_specifiers = {
...     'Violin Voice': None,
...     'Cello Voice': None,
... }

>>> target_timespan = timespantools.Timespan((1, 2), (2, 1))

>>> timespan_inventory = timespan_maker(
...     music_specifiers=music_specifiers,
...     target_timespan=target_timespan,
...     timespan_inventory=timespan_inventory,
... )

>>> print(format(timespan_inventory))
	timespantools.TimespanInventory(
	    [
	        consort.tools.PerformedTimespan(
	            start_offset=durationtools.Offset(1, 4),
	            stop_offset=durationtools.Offset(1, 1),
	            voice_name='Viola Voice',
	        ),
	        consort.tools.PerformedTimespan(
	            start_offset=durationtools.Offset(1, 2),
	            stop_offset=durationtools.Offset(3, 4),
	            voice_name='Cello Voice',
	        ),
	        consort.tools.PerformedTimespan(
	            start_offset=durationtools.Offset(1, 2),
	            stop_offset=durationtools.Offset(3, 4),
	            voice_name='Violin Voice',
	        ),
	        consort.tools.PerformedTimespan(
	            start_offset=durationtools.Offset(3, 4),
	            stop_offset=durationtools.Offset(1, 1),
	            voice_name='Cello Voice',
	        ),
	        consort.tools.PerformedTimespan(
	            start_offset=durationtools.Offset(3, 4),
	            stop_offset=durationtools.Offset(1, 1),
	            voice_name='Violin Voice',
	        ),
	    ]

497
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(1, 1),
    stop_offset=durationtools.Offset(3, 2),
    voice_name='Viola Voice',
),
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(1, 1),
    stop_offset=durationtools.Offset(3, 2),
    voice_name='Cello Voice',
),
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(1, 1),
    stop_offset=durationtools.Offset(3, 2),
    voice_name='Violin Voice',
)

### CLASS VARIABLES ###

__slots__ = ('_hysteresis',
              '_include_inner_starts',
              '_include_inner_stops',
              '_inspect_music',
              '_labels',
              '_rotation_indices',
              '_voice_names',
)

### INITIALIZER ###

def __init__(
    self,
    hysteresis=None,
    include_inner_starts=None,
    include_inner_stops=None,
    inspect_music=None,
    labels=None,
    output_masks=None,
    padding=None,
    rotation_indices=None,
    seed=None,
    timespanSpecifier=None,
    voice_names=None,
):  
    TimespanMaker.__init__(
        self,
        output_masks=output_masks,
        padding=padding,
        seed=seed,
        timespanSpecifier=timespanSpecifier,
    )
if hysteresis is not None:
    hysteresis = durationtools.Duration(hysteresis)
    assert 0 < hysteresis
self._hysteresis = hysteresis
if include_inner_starts is not None:
    include_inner_starts = bool(include_inner_starts)
self._include_inner_starts = include_inner_starts
if include_inner_stops is not None:
    include_inner_stops = bool(include_inner_stops)
self._include_inner_stops = include_inner_stops
if inspect_music is not None:
    inspect_music = bool(inspect_music)
self._inspect_music = inspect_music
if rotation_indices is not None:
    if not isinstance(rotation_indices, collections.Sequence):
        rotation_indices = int(rotation_indices)
    rotation_indices = (rotation_indices,)
self._rotation_indices = rotation_indices
if labels is not None:
    if isinstance(labels, str):
        labels = (labels,)
    labels = tuple(str(_) for _ in labels)
self._labels = labels
if voice_names is not None:
    voice_names = tuple(voice_names)
self._voice_names = voice_names

### PRIVATE METHODS ###

def _collect_preexisting_timespans(
    self,
    target_timespan=None,
    timespan_inventory=None,
):
    import consort
timespan_inventory = timespantools.TimespanInventory()
for timespan in timespan_inventory:
    if not isinstance(timespan, consort.PerformedTimespan):
        continue
    if (self.voice_names and
        timespan.voice_name not in self.voice_names):
        continue
    if not self.labels:
        pass
    elif not hasattr(timespan, 'music_specifier') or |
        not timespan.music_specifier or |
        not timespan.music_specifier.labels:
        continue
    elif not any(label in timespan.music_specifier.labels
            for label in self.labels):
        continue
preexisting_timespans.append(timespan)

if self.inspect_music and timespan.music:
    outer_start_offset = timespan.start_offset
    inner_start_offset = inspect_(timespan.music).get_timespan().start_offset
    assert inner_start_offset == 0

    for division in timespan.music:
        division_timespan = inspect_(division).get_timespan().translate(
            outer_start_offset)

        preexisting_timespans.append(division_timespan)

return preexisting_timespans & target_timespan

def _partition_preexisting_timespans(self, timespans):
    shards = timespans.partition(include_tangent_timespans=True)
    if not self.hysteresis or not shards:
        return shards

    coalesced_shards = [shards[0]]
    for shard in shards[1:]:
        last_stop = coalesced_shards[-1].stop_offset
        this_start = shard.start_offset
        gap = this_start - last_stop

        if self.hysteresis <= gap:
            coalesced_shards.append(shard)
        else:
            coalesced_shards[-1].extend(shard)
            coalesced_shards[-1].sort()

    return coalesced_shards

def _make_timespans(
    self,
    layer=None,
    music_specifiers=None,
    target_timespan=None,
    timespan_inventory=None,
):
    new_timespans = timespantools.TimespanInventory()
    if not self.voice_names and not self.labels:
        return new_timespans

    rotation_indices = self.rotation_indices or (0,)
    rotation_indices = datastructuretools.CyclicTuple(rotation_indices)
    context_counter = collections.Counter()
    preexisting_timespans = self._collect_preexisting_timespans(
        target_timespan=target_timespan,
        timespan_inventory=timespan_inventory,
    )

    partitioned_timespans = self._partition_preexisting_timespans(
        preexisting_timespans)

    for group_index, group in enumerate(partitioned_timespans):
        rotation_index = rotation_indices[group_index]
        offsets = set()
        offsets.add(group.start_offset)
        offsets.add(group.stop_offset)
for timespan in group:
    if self.include_inner_starts:
        offsets.add(timespan.start_offset)
    if self.include_inner_stops:
        offsets.add(timespan.stop_offset)
    offsets = tuple(sorted(offsets))
    durations = mathtools.difference_series(offsets)
    durations = sequencetools.rotate_sequence(durations, rotation_index)
    start_offset = offsets[0]
    for context_name, music_specifier in music_specifiers.items():
        context_seed = context_counter[context_name]
        timespans = music_specifier(
            durations=durations,
            layer=layer,
            output_masks=self.output_masks,
            padding=self.padding,
            seed=context_seed,
            start_offset=start_offset,
            timespan_specifier=self.timespan_specifier,
            voice_name=context_name,
        )
        context_counter[context_name] += 1
        new_timespans.extend(timespans)
    return new_timespans

### PUBLIC PROPERTIES ###

@property
def hysteresis(self):
    return self._hysteresis

@property
def include_inner_starts(self):
    return self._include_inner_starts

@property
def include_inner_stops(self):
    return self._include_inner_stops

@property
def inspect_music(self):
    return self._inspect_music

@property
def is_dependent(self):
    return True

@property
def labels(self):
    return self._labels

@property
def rotation_indices(self):
A.19  consort.tools.DynamicExpression

>>> import consort
>>> dynamic_expression = consort.DynamicExpression(
...     dynamic_tokens='f p pp pp',
...     transitions=('flared', None),
... )
>>> print(format(dynamic_expression))
consort.tools.DynamicExpression(
    dynamic_tokens=datastructuretools.CyclicTuple(
        ['f', 'p', 'pp', 'pp'],
    ),
    transitions=datastructuretools.CyclicTuple(
        ['flared', None],
    ),
)

>>> music = Staff(r'''
    { c'4 d'4 e'4 f'4 }
    ... { g'4 a'4 b'4 }
    ... { c''4 }
    ... '}
    >>> print(format(music))
\new Staff {
    { 
        c'4 
        d'4 
        e'4 
        f'4 
    } 
    { 
        g'4 
        a'4 
        b'4 
    } 
    { 
        c''4 
    } 
} 
:: 

>>> dynamic_expression(music)
>>> print(format(music))
\new Staff {
    { 
        \once \override Hairpin.stencil = #flared-hairpin 
        c'4 \f \>
        d'4 
        e'4 
        f'4 
    } 
    { 
        g'4 \p \>
        a'4 
        b'4 
    } 
    { 
        c''4 \pp 
    } 
} 
.. container:: example 
:: 

>>> music = Staff(r''' ...
    ... { c'4 d'4 e'4 f'4 } 
    ... { g'4 a'4 b'4 } 
    ... { c''4 c'4 } 
    ... '''
:: 

>>> dynamic_expression(music, seed=2)
>>> print(format(music))
\new Staff {
```
.. container:: example
::

>>> music = Staff("{ c'4 }")
>>> dynamic_expression(music, seed=1)
>>> print(format(music))
\new Staff {
    { c'4 \p }
}

.. container:: example
::

>>> music = Staff("{ c'4 d'4 }")
>>> dynamic_expression(music, seed=1)
>>> print(format(music))
\new Staff {
    { c'4 \p \\
      d'4 \pp }
}

.. container:: example
::

>>> music = Staff("{ r4 c'4 } { r4 d'4 r4 } { r4 e' r4 } ")
>>> dynamic_expression(music)
>>> print(format(music))
\new Staff {
    {
```
\once \override Hairpin.stencil = #flared-hairpin
c'4 \f \\
r4
)
{
  r4
d'4 \p \\
r4
}
{
  r4
e'4 \pp
}

.. container:: example

::

>>> music = Staff("{ c'16 c'16 }")
>>> dynamic_expression(music)
>>> print(format(music))
\new Staff {
  c'16 \f
c'16
}

.. container:: example

::

>>> music = Staff("{ c'1 }")
>>> dynamic_expression = consort.DynamicExpression(
  ...  dynamic_tokens='fp',
  ...  )
>>> dynamic_expression(music)
>>> print(format(music))
\new Staff {
  c'1 \fp
}

.. container:: example

::

>>> music = Staff("{ c'1 }")
>>> dynamic_expression = consort.DynamicExpression(
  ...  dynamic_tokens='fp',
  ...  )
... unsustained=True,
...

>>> dynamic_expression(music)

>>> print(format(music))
\new Staff {
  
  \once \override Hairpin.circled-tip = ##t
c'4 \<
d'4
e'4
  }

  
  \once \override Hairpin.circled-tip = ##t
c'4 \p \>
d'4
e'4
  }

  
  \once \override Hairpin.circled-tip = ##t
c'4 \p
d'4
e'4
  }

  
  \once \override Hairpin.circled-tip = ##t
c'4
d'4
e'4
  }

  \once \override Hairpin.circled-tip = ##t
c'4
d'4
e'4
  }

  
  \once \override Hairpin.circled-tip = ##t
c'4
d'4
e'4
  }

  \once \override Hairpin.circled-tip = ##t
c'4
d'4
e'4
  }

  \once \override Hairpin.circled-tip = ##t
c'4
c'4
}

.. container:: example

::

>>> music = Staff(r''
... { c'4 d'4 e'4 }
... { c'4 d'4 e'4 }
... { c'4 d'4 e'4 }
... { c'4 d'4 e'4 }
... { c'4 d'4 e'4 }
... ''')

>>> dynamic_expression = consort.DynamicExpression(
... division_period=2,
... dynamic_tokens='p ppp',
... start_dynamic_tokens='o',
... stop_dynamic_tokens='o',
... )

>>> dynamic_expression(music)

>>> print(format(music))
\new Staff {
  
  \once \override Hairpin.circled-tip = ##t
c'4 \<
d'4
e'4
  }

  
  \once \override Hairpin.circled-tip = ##t
c'4 \p \>
d'4
e'4
  }

  
  \once \override Hairpin.circled-tip = ##t
c'4
d'4
e'4
  }

  
  \once \override Hairpin.circled-tip = ##t
c'4
d'4
e'4
  }

  
  \once \override Hairpin.circled-tip = ##t
c'4
d'4
e'4
  }

  
  \once \override Hairpin.circled-tip = ##t
c'4
c'4
}
>>> music = Staff("{ c'8. } { e'8. } { g'8. }")
>>> dynamic_expression = consort.DynamicExpression(
... division_period=2,
... dynamic_tokens='p ppp',
... start_dynamic_tokens='o',
... stop_dynamic_tokens='o',
... )
>>> dynamic_expression(music)
>>> print(format(music))
\new Staff {
  \once \override Hairpin.circled-tip = ##t
  c'8. <
  }
  e'8.
  }
  g'8. \p
  }
  \tweak #'text #tuplet-number::calc-fraction-text
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }

>>> music = Staff(r'"
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }

\new Staff { 
  \once \override Hairpin.circled-tip = ##t
  c'8. <
  }
  e'8.
  }
  g'8. \p
  }
  \tweak #'text #tuplet-number::calc-fraction-text
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }

\new Staff { 
  \once \override Hairpin.circled-tip = ##t
  c'8. <
  }
  e'8.
  }
  g'8. \p
  }
  \tweak #'text #tuplet-number::calc-fraction-text
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }

\new Staff { 
  \once \override Hairpin.circled-tip = ##t
  c'8. <
  }
  e'8.
  }
  g'8. \p
  }
  \tweak #'text #tuplet-number::calc-fraction-text
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }
  }
  \times 3/4 { d'16 d' d' r d' d' r }

```python
### CLASS VARIABLES ###
__slots__ = ('_division_period', '_dynamic_tokens', '_only_first', '_start_dynamic_tokens', '_stop_dynamic_tokens', '_transitions', '_unsustained',)

_transition_types = ('constante', 'flared', 'simple', None)

### INITIALIZER ###
def __init__(
    self,
    dynamic_tokens=('ppp',),
    division_period=None,
    only_first=None,
    start_dynamic_tokens=None,
    stop_dynamic_tokens=None,
    transitions=None,
    unsustained=None,
):
    dynamic_tokens = self._tokens_to_cyclic_tuple(dynamic_tokens)
    assert dynamic_tokens
    self._dynamic_tokens = dynamic_tokens
    if division_period is not None:
        division_period = int(division_period)
        assert 0 < division_period
    self._division_period = division_period
    self._start_dynamic_tokens = self._tokens_to_cyclic_tuple(start_dynamic_tokens)
    self._stop_dynamic_tokens = self._tokens_to_cyclic_tuple(stop_dynamic_tokens)
```
if isinstance(transitions, (str, type(None))):
    transitions = [transitions]
assert all(_ in self._transition_types for _ in transitions)
transitions = datastructuretools.CyclicTuple(transitions)
self._transitions = transitions
if only_first is not None:
    only_first = bool(only_first)
sel._only_first = only_first
if unsustained is not None:
    unsustained = bool(unsustained)
sel._unsustained = unsustained

### SPECIAL METHODS ###

def __call__(self, music, name=None, seed=0):
    original_seed = seed
    current_dynamic = None
    current_hairpin = None
    selections, components = sel._get_selections(music)
    #print(selections)
    #print(components)
    length = len(components)
    if sel.only_first:
        length = 1
    components = components[:1]
    for index, component in enumerate(components[:length]):
        selection = selections[index]
        dynamic, hairpin, hairpin_override = sel._get_attachments(
            index, length, seed, original_seed)
        if dynamic != current_dynamic:
            if dynamic.name != 'o':
                attach(dynamic, component, name=name)
            current_dynamic = dynamic
        if sel.unsustained:
            inner_leaves = selection[1:-1]
            prototype = scoretools.Rest
            if (len(inner_leaves) and
                all(isinstance(_, prototype) for _ in inner_leaves)):
                hairpin = None
            if hairpin is not None:
                attach(hairpin, selection, name=name)
            current_hairpin = hairpin
        if current_hairpin is not None and hairpin_override is not None:
            attach(hairpin_override, component, name=name)
        seed += 1
        dynamic, _, _ = sel._get_attachments(
            length - 1, length, seed, original_seed)
    if sel.unsustained:
        if dynamic is not None:
            if length == 1:
                if not selections or len(selections[0]) < 4:
                    if dynamic.name in dynamic._composite_dynamic_name_to_steady_state_dynamic_name:
dynamic_name = dynamic._composite_dynamic_name_to_steady_state_dynamic_name[dynamic.name]

dynamic = indicator_tools.Dynamic(dynamic_name)

if dynamic != current_dynamic and dynamic.name != 'o':
    attach(dynamic, components[-1], name=name)

if dynamic.name == 'o' and current_hairpin:
    next_leaf = components[-1]._get_leaf()  
    if next_leaf is not None:
        current_hairpin._append(next_leaf)

### PRIVATE METHODS ###

def _get_attachments(self, index, length, seed, original_seed):
    dynamic_seed = seed

    if self.start_dynamic_tokens:
        dynamic_seed -= 1

    this_token = None
    next_token = None
    this_dynamic = None
    next_dynamic = None
    hairpin = None
    hairpin_override = None

    if length == 1:
        if self.start_dynamic_tokens:
            this_token = self.start_dynamic_tokens[original_seed]
        elif self.stop_dynamic_tokens:
            this_token = self.stop_dynamic_tokens[original_seed]
        else:
            this_token = self.dynamic_tokens[dynamic_seed]
        if this_token == 'o':
            this_token = self.dynamic_tokens[dynamic_seed]
        else:
            this_token = self.dynamic_tokens[dynamic_seed + 1]
    elif length == 2:
        if index == 0:
            if self.start_dynamic_tokens:
                this_token = self.start_dynamic_tokens[original_seed]
            else:
                this_token = self.dynamic_tokens[dynamic_seed]
        elif self.stop_dynamic_tokens:
            next_token = self.stop_dynamic_tokens[original_seed]
        else:
            next_token = self.dynamic_tokens[dynamic_seed + 1]
        if next_token == 'o':
            this_token = self.dynamic_tokens[dynamic_seed]
        else:
            this_token = self.dynamic_tokens[dynamic_seed]
```python
next_token = self.dynamic_tokens[dynamic_seed]
else:
    #print('!!!', index)
    if index == 0:
        if self.start_dynamic_tokens:
            this_token = self.start_dynamic_tokens[original_seed]
            next_token = self.dynamic_tokens[dynamic_seed + 1]
            #print('A1', this_token, next_token)
        else:
            this_token = self.dynamic_tokens[dynamic_seed]
            next_token = self.dynamic_tokens[dynamic_seed + 1]
            #print('A2', this_token, next_token)
    elif index == length - 1:  # Last component.
        if self.stop_dynamic_tokens:
            this_token = self.stop_dynamic_tokens[original_seed]
            #print('B1', this_token, next_token)
        else:
            this_token = self.dynamic_tokens[dynamic_seed]
            #print('B2', this_token, next_token)
    elif index == length - 2:  # Next to last component.
        this_token = self.dynamic_tokens[dynamic_seed]
        if self.stop_dynamic_tokens:
            next_token = self.stop_dynamic_tokens[original_seed]
            #print('C1', this_token, next_token)
        else:
            next_token = self.dynamic_tokens[dynamic_seed + 1]
            #print('C2', this_token, next_token)
    else:
        this_token = self.dynamic_tokens[dynamic_seed]
        next_token = self.dynamic_tokens[dynamic_seed + 1]
        #print('D1', this_token, next_token)

this_dynamic = indicatortools.Dynamic(this_token)
this_dynamic_ordinal = mathtools.NegativeInfinity()
if this_dynamic.name != 'o':
    this_dynamic_ordinal = this_dynamic.ordinal
if next_token is not None:
    next_dynamic = indicatortools.Dynamic(next_token)
    next_dynamic_ordinal = mathtools.NegativeInfinity()
    if next_dynamic.name != 'o':
        next_dynamic_ordinal = next_dynamic.ordinal

if next_dynamic is not None:
    if this_dynamic.ordinal < next_dynamic.ordinal:
        hairpin = spannertools.Crescendo(include_rests=True)
    elif next_dynamic.ordinal < this_dynamic.ordinal:
        hairpin = spannertools.Decrescendo(include_rests=True)
if hairpin is not None:
    transition = self.transitions[seed]
    if transition == 'constante':
        hairpin = spannertools.Crescendo(include_rests=True)
    if transition in ('flared', 'constante'):
        hairpin_override = lilypondnametools.LilyPondGrobOverride(
```

grob_name='Hairpin',
        is_once=True,
        property_path='stencil',
        value=schemetools.Scheme('{}-hairpin'.format(transition)),
    )
    if this_dynamic.name == 'o' or next_dynamic.name == 'o':
        hairpin_override = lilypondnametools.LilyPondGrobOverride(
            grob_name='Hairpin',
            is_once=True,
            property_path='circled-tip',
            value=True,
        )
        #print(index, this_dynamic, next_dynamic, hairpin)
return this_dynamic, hairpin, hairpin_override

def _partition_selections(self, music):
    period = self.division_period or 1
    selections = [_.select_leaves() for _ in music]
    parts = sequencetools.partition_sequence_by_counts(
        selections, [period], cyclic=True, overhang=True)
    if len(parts[-1]) < period and 1 < len(parts):
        part = parts.pop()
        parts[-1].extend(part)
    selections = []
    for part in parts:
        selection = part[0]
        for next_selection in part[1:]:
            selection = selection + next_selection
        selections.append(selection)
    return selections

def _reorganize_selections(self, selections):
    prototype = (scoretools.Note, scoretools.Chord)
    for i, leaf in enumerate(selections[0]):
        if isinstance(leaf, prototype):
            break
    selections[0] = selections[0][i:]
    for i, leaf in enumerate(reversed(selections[-1])):
        if isinstance(leaf, prototype):
            break
    if i == 0:
        i = None
    else:
        i = -i
    selections[-1] = selections[-1][i:
    if len(selections) == 1:
        return selections
    for i in range(len(selections) - 1):
        selection_one, selection_two = selections[i], selections[i + 1]
        for j, leaf in enumerate(selection_two):
            if isinstance(leaf, prototype):
                break
if 0 < j:
    left, right = selection_two[:j], selection_two[j:]
    selection_one = selection_one + left
    selection_two = right
    selections[i] = selection_one
    selections[i + 1] = selection_two
return selections

def _get_selections(self, music):
    """Gets selections and attach components from 'music'.
    ""
    .. container:: example

    ::

        >>> music = Staff(r''
        ...             { c'4 d'4 e'4 f'4 }
        ...             { g'4 a'4 b'4 }
        ...             { c''4 }
        ...         '')
        >>> dynamic_expression = consort.DynamicExpression("f")
        >>> result = dynamic_expression._get_selections(music)
        >>> selections, attach_components = result
        >>> for _ in selections:
        ...     _
        ...     ContiguousSelection(Note("c'4"), Note("d'4"), Note("e'4"), Note("f'4"), Note("g'4"))
        ContiguousSelection(Note("g'4"), Note("a'4"), Note("b'4"), Note("c''4"))

    ::

        >>> for _ in attach_components:
        ...     _
        ...     Note("c'4")
        Note("g'4")
        Note("c''4")

    .. container:: example

    ::

        >>> music = Staff(r''
        ...             { c'4 d'4 e'4 }
        ...             { f'4 g'4 a'4 }
        ...             { b'4 c''4 }
        ...         '')
        >>> dynamic_expression = consort.DynamicExpression("f")
        >>> result = dynamic_expression._get_selections(music)
        >>> selections, attach_components = result
        >>> for _ in selections:
        ...     _
        ...     ContiguousSelection(Note("c'4"), Note("d'4"), Note("e'4"), Note("f'4"))
ContiguousSelection(Note("f'4"), Note("g'4"), Note("a'4"), Note("b'4"))
ContiguousSelection(Note("b'4"), Note("c'4"))

::

>>> for _ in attach_components:
    ...
    ...
    Note("c'4")
    Note("f'4")
    Note("b'4")
    Note("c'4")

.. container:: example

::

>>> music = Staff(r''
... { c'8 d'8 e'8 }
... { f'8 g'8 a'8 }
... { b'32 c''16. }
... ''

>>> result = dynamic_expression._get_selections(music)
>>> selections, attach_components = result

>>> for _ in selections:
    ...
    ...
    ContiguousSelection(Note("c'8"), Note("d'8"), Note("e'8"), Note("f'8"))
    ContiguousSelection(Note("f'8"), Note("g'8"), Note("a'8"), Note("b'32"), Note("c'16."))

::

>>> for _ in attach_components:
    ...
    ...
    Note("c'8")
    Note("f'8")
    Note("c'16.")

.. container:: example

::

>>> music = Staff("{ r4 c'4 r4 } { r4 d'4 r4 } { r4 e' r4 } ")

>>> result = dynamic_expression._get_selections(music)

>>> selections, attach_components = result

>>> for _ in selections:
    ...
    ...
    ContiguousSelection(Note("c'4"), Rest('r4'), Rest('r4'), Note("d'4"))
    ContiguousSelection(Note("d'4"), Rest('r4'), Rest('r4'), Note("e'4"))
>>> for _ in attach_components:
... _
... _
... Note("c'4")
... Note("d'4")
... Note("e'4")

.. container:: example

    ::

    >>> music = Staff("{ c'8. } { e'8. } { g'8. }")
    >>> dynamic_expression = consort.DynamicExpression(
    ...   division_period=2,
    ...   dynamic_tokens='p ppp',
    ...   start_dynamic_tokens='o',
    ...   stop_dynamic_tokens='o',
    ...)
    >>> result = dynamic_expression._get_selections(music)
    >>> selections, attach_components = result
    >>> for _ in selections:
    ... _
    ... _
    ... ContiguousSelection(Note("c'8."), Note("e'8."), Note("g'8."))

    ::

    >>> for _ in attach_components:
    ... _
    ... _
    ... Note("c'8.")
    ... Note("g'8.")

    ""
    #print('---', music)
    initial_selections = self._partition_selections(music)
    #print('  ', initial_selections)
    initial_selections = self._reorganize_selections(initial_selections)
    #print('  ', initial_selections)
    attach_components = []
    selections = []
    assert len(initial_selections)
    for i, selection in enumerate(initial_selections):
        #print('  ', i, selection)
        if i < len(initial_selections) - 1:
            #print('   ', 'A')
            selection = selection + (initial_selections[i + 1][0],)
            selections.append(selection)
            attach_components.append(selection[0])
        elif ((selection.get_duration() <= durationtools.Duration(1, 8) and
               1 < len(selections)) or len(selection) == 1):
            #print('   ', 'B')
            attach_components.append(selection[-1])
if selections:
    #print(' ', 'B1')
    selections[-1] = selections[-1] + selection[1:]  #print(' ', 'B1')
    elif (
        durationtools.Duration(1, 8) < (selection[-1]._get_timespan().start_offset -
        selection[0]._get_timespan().start_offset
    ) :
        #print(' ', 'C')
        selections.append(selection)
        attach_components.append(selection[0])
        attach_components.append(selection[-1])
else:
    #print(' ', 'D')
    attach_components.append(selection[0])
    #print(' ', initial_selections)
    #print(' ', attach_components)
return selections, attach_components

def _tokens_to_cyclic_tuple(self, tokens):
    if tokens is None:
        return tokens
    if isinstance(tokens, str):
        tokens = tokens.split()
    for token in tokens:
        if token == 'o':
            continue
        assert token in indicatortools.Dynamic._dynamic_names
    assert len(tokens)
    tokens = datastructuretools.CyclicTuple(tokens)
    return tokens

### PUBLIC PROPERTIES ###

@property
def division_period(self):
    return self._division_period

@property
def dynamic_tokens(self):
    return self._dynamic_tokens

@property
def only_first(self):
    return self._only_first

@property
def period(self):
    return self._period

@property
def start_dynamic_tokens(self):
    return self._start_dynamic_tokens
A.20 consort.tools.FloodedTimespanMaker

```python
# -*- encoding: utf-8 -*-
from abjad.tools import timespantools
from consort.tools.TimespanMaker import TimespanMaker

class FloodedTimespanMaker(TimespanMaker):
    r'''
    A flooded timespan maker.
    
    >>> import consort
    >>> timespan_maker = consort.FloodedTimespanMaker()
    >>> print(format(timespan_maker))
    consort.tools.FloodedTimespanMaker()
    
    >>> music_specifiers = {
    ...     'Violin Voice': 'violin music',
    ...     'Cello Voice': 'cello music',
    ...     }
    >>> target_timespan = timespantools.Timespan((1, 2), (2, 1))
    >>> timespan_inventory = timespan_maker(
    ...     music_specifiers=music_specifiers,
    ...     target_timespan=target_timespan,
    ...     )
    >>> print(format(timespan_inventory))
    timespantools.TimespanInventory([
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(1, 2),
            stop_offset=durationtools.Offset(2, 1),
            music_specifier='cello music',
            voice_name='Cello Voice',
        ),
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(1, 2),
            stop_offset=durationtools.Offset(2, 1),
            music_specifier='violin music',
        ),
    ])
```
```python
>>> music_specifier = consort.CompositeMusicSpecifier(
...     primary_music_specifier='one',
...     primary_voice_name='Viola 1 RH',
...     rotation_indices=(0, 1, -1),
...     secondary_voice_name='Viola 1 LH',
...     secondary_music_specifier=consort.MusicSpecifierSequence(
...         application_rate='phrase',
...         music_specifiers=['two', 'three', 'four'],
...     ),
... )
>>> music_specifiers = {
...     'Viola 1 Performer Group': music_specifier,
... }
>>> timespan_inventory = timespan_maker(
...     music_specifiers=music_specifiers,
...     target_timespan=target_timespan,
... )
>>> print(format(timespan_inventory))
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(1, 2),
    stop_offset=durationtools.Offset(2, 1),
    music_specifier='two',
    voice_name='Viola 1 LH',
),
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(1, 2),
    stop_offset=durationtools.Offset(2, 1),
    music_specifier='one',
    voice_name='Viola 1 RH',
),
```

```python
### CLASS VARIABLES ###

```python
__slots__ = ()
```

```python
### INITIALIZER ###

```python
def __init__(
    self,
    output_masks=None,
    padding=None,
```


```python
def _make_timespans(self, layer=None, music_specifiers=None, target_timespan=None, timespan_inventory=None):
    start_offset = target_timespan.start_offset
    durations = [target_timespan.duration]
    new_timespans = timespantools.TimespanInventory()
    for context_name, musicSpecifier in music_specifiers.items():
        timespans = musicSpecifier(
            durations=durations,
            layer=layer,
            output_masks=self.output_masks,
            padding=self.padding,
            seed=self.seed,
            start_offset=start_offset,
            timespanSpecifier=self.timespanSpecifier,
            voice_name=context_name,
        )
        new_timespans.extend(timespans)
    return new_timespans
```

---

### PUBLIC METHODS ###

```python
@classmethod
def make_timespan(cls, start_offset, duration, layer=None, output_masks=None, padding=None, seed=None, timespanSpecifier=None, voice_name=None):
    timespan = abjad.Timespan(start_offset, start_offset + duration)
    attach(voice_name=voice_name)
    attach(rotation=1)
    attach(opacity=1)
    attach(layer=layer)
    attach(output_masks=output_masks)
    attach(padding=padding)
    attach(seed=seed)
    attach(timespanSpecifier=timespanSpecifier)
    attach(opacity=1)
    attach(rotation=1)
    attach(layer=layer)
    attach(output_masks=output_masks)
    attach(padding=padding)
    attach(seed=seed)
    attach(timespanSpecifier=timespanSpecifier)
```

---

### PRIVATE METHODS ###

```python
def _make_timespans(self, layer=None, music_specifiers=None, target_timespan=None, timespan_inventory=None):
    start_offset = target_timespan.start_offset
    durations = [target_timespan.duration]
    new_timespans = timespantools.TimespanInventory()
    for context_name, musicSpecifier in music_specifiers.items():
        timespans = musicSpecifier(
            durations=durations,
            layer=layer,
            output_masks=self.output_masks,
            padding=self.padding,
            seed=self.seed,
            start_offset=start_offset,
            timespanSpecifier=self.timespanSpecifier,
            voice_name=context_name,
        )
        new_timespans.extend(timespans)
    return new_timespans
```

---

### PUBLIC METHODS ###

```python
@classmethod
def make_timespans(cls, start_offsets, durations, layer=None, output_masks=None, padding=None, seed=None, timespanSpecifier=None, voice_name=None):
    timespans = []
    for i in range(len(start_offsets)):
        start_offset = start_offsets[i]
        duration = durations[i]
        timespan = abjad.Timespan(start_offset, start_offset + duration)
        timespans.append(timespan)
    return abjad.TimespanCollection(timespans)
```

---

### PRIVATE METHODS ###

```python
def _timespans_to_timespan_collection(self):
    timespans = []
    for timespan in self:
        timespans.append(timespan)
    return abjad.TimespanCollection(timespans)
```

---

### PUBLIC METHODS ###

```python
@classmethod
def make_timespans_collection(cls, start_offsets, durations, layer=None, output_masks=None, padding=None, seed=None, timespanSpecifier=None, voice_name=None):
    timespans = []
    for i in range(len(start_offsets)):
        start_offset = start_offsets[i]
        duration = durations[i]
        timespan = abjad.Timespan(start_offset, start_offset + duration)
        timespans.append(timespan)
    return abjad.TimespanCollection(timespans)
```

---

### PRIVATE METHODS ###

```python
def _timespans_to_timespan_collection(self):
    timespans = []
    for timespan in self:
        timespans.append(timespan)
    return abjad.TimespanCollection(timespans)
```

---

### PUBLIC METHODS ###

```python
@classmethod
def make_timespans_collection(cls, start_offsets, durations, layer=None, output_masks=None, padding=None, seed=None, timespanSpecifier=None, voice_name=None):
    timespans = []
    for i in range(len(start_offsets)):
        start_offset = start_offsets[i]
        duration = durations[i]
        timespan = abjad.Timespan(start_offset, start_offset + duration)
        timespans.append(timespan)
    return abjad.TimespanCollection(timespans)
```

---

### PRIVATE METHODS ###

```python
def _timespans_to_timespan_collection(self):
    timespans = []
    for timespan in self:
        timespans.append(timespan)
    return abjad.TimespanCollection(timespans)
```

---

### PUBLIC METHODS ###

```python
@classmethod
def make_timespans_collection(cls, start_offsets, durations, layer=None, output_masks=None, padding=None, seed=None, timespanSpecifier=None, voice_name=None):
    timespans = []
    for i in range(len(startOffsets)):
        start_offset = startOffsets[i]
        duration = durations[i]
        timespan = abjad.Timespan(start_offset, start_offset + duration)
        timespans.append(timespan)
    return abjad.TimespanCollection(timespans)
```

---

### PRIVATE METHODS ###

```python
def _timespans_to_timespan_collection(self):
    timespans = []
    for timespan in self:
        timespans.append(timespan)
    return abjad.TimespanCollection(timespans)
```

---

### PUBLIC METHODS ###

```python
@classmethod
def make_timespans_collection(cls, start_offsets, durations, layer=None, output_masks=None, padding=None, seed=None, timespanSpecifier=None, voice_name=None):
    timespans = []
    for i in range(len(start_offsets)):
        start_offset = start_offsets[i]
        duration = durations[i]
        timespan = abjad.Timespan(start_offset, start_offset + duration)
        timespans.append(timespan)
    return abjad.TimespanCollection(timespans)
```

---

### PRIVATE METHODS ###

```python
def _timespans_to_timespan_collection(self):
    timespans = []
    for timespan in self:
        timespans.append(timespan)
    return abjad.TimespanCollection(timespans)
```
A grace maker.

>>> import consort
>>> grace_handler = consort.GraceHandler(
...     counts=(0, 1, 0, 0, 2),
...     )
>>> print(format(grace_handler))
consort.tools.GraceHandler(
    counts=datastructuretools.CyclicTuple([0, 1, 0, 0, 2]),
)

### CLASS VARIABLES ###

__slots__ = ('_counts',
             '_only_if_preceded_by_nonsilence',
             '_only_if_preceded_by_silence',
)

### INITIALIZER ###

def __init__(
    self,
    counts=(1,),
    only_if_preceded_by_nonsilence=None,
    only_if_preceded_by_silence=None,
):
    if not counts:
        counts = (0,)
    if not isinstance(counts, collections.Sequence):
        counts = (counts,)
    assert len(counts)
    assert mathtools.all_are_nonnegative_integer_equivalent_numbers(counts)
    self._counts = datastructuretools.CyclicTuple(counts)
    if only_if_preceded_by_nonsilence is not None:
        only_if_preceded_by_nonsilence = bool(only_if_preceded_by_nonsilence)
    if only_if_preceded_by_silence is not None:
        only_if_preceded_by_silence = bool(only_if_preceded_by_silence)
    assert not (only_if_preceded_by_silence and only_if_preceded_by_nonsilence)
    self._only_if_preceded_by_silence = only_if_preceded_by_silence
    self._only_if_preceded_by_nonsilence = only_if_preceded_by_nonsilence

### SPECIAL METHODS ###
def __call__(
    self,
    logical_tie,
    seed=0,
):
    assert isinstance(logical_tie, selectiontools.LogicalTie)
    if self.counts is None:
        return
    previous_leaf = logical_tie.head._get_leaf(-1)
    if previous_leaf is None:
        return
    silence_prototype = (
        scoretools.Rest,
        scoretools.MultimeasureRest,
        scoretools.Skip,
    )
    if self.only_if_preceded_by_silence:
        if not isinstance(previous_leaf, silence_prototype):
            return
    if self.only_if_preceded_by_nonsilence:
        if isinstance(previous_leaf, silence_prototype):
            return
    grace_count = self.counts[seed]
    if not grace_count:
        return
    kind = 'after'
    leaf_to_attach_to = previous_leaf
    leaves = []
    notes = scoretools.make_notes([0], [(1, 16)] * grace_count)
    leaves.extend(notes)
    assert len(leaves)
    grace_container = scoretools.GraceContainer(
        leaves,
        kind=kind,
    )
    override(grace_container).flag.stroke_style = '\n' + schemetools.Scheme('grace', force_quotes=True)
    override(grace_container).script.font_size = 0.5
    attach(grace_container, leaf_to_attach_to)

### PRIVATE METHODS ###

@staticmethod
def _process_session(segment_maker):
    import consort
    counter = collections.Counter()
    attack_point_map = segment_maker.attack_point_map
    for logical_tie in attack_point_map:
        musicSpecifier = '\n' + consort.SegmentMaker.logical_tie_to_musicSpecifier(
            logical_tie)
        if not musicSpecifier:
            continue
grace_handler = musicSpecifier.grace_handler
if not grace_handler:
    continue
previous_leaf = logical_tie.head._get_leaf(-1)
if previous_leaf is None:
    continue
if musicSpecifier not in counter:
    seed = musicSpecifier.seed or 0
    counter[musicSpecifier] = seed
    seed = counter[musicSpecifier]
    grace_handler(
        logical_tie,
        seed=seed,
    )
    counter[musicSpecifier] += 1

### PUBLIC METHODS ###

def reverse(self):
    counts = self.counts
    if counts is not None:
        counts = counts.reverse()
    return new(self,
        counts=counts,
    )

def rotate(self, n=0):
    counts = self.counts
    if counts is not None:
        counts = counts.rotate(n)
    return new(self,
        counts=counts,
    )

### PUBLIC PROPERTIES ###

@property
def counts(self):
    return self._counts

@property
def only_if_preceded_by_nonsilence(self):
    return self._only_if_preceded_by_nonsilence

@property
def only_if_preceded_by_silence(self):
    return self._only_if_preceded_by_silence

A.22 consort.tools.HarmonicExpression

# -*- encoding: utf-8 -*-
from abjad.tools import pitchtools
from abjad.tools import scoretools
from consort.tools.LogicalTieExpression import LogicalTieExpression
class HarmonicExpression(LogicalTieExpression):
   r'''A harmonic expression.
   ::
>>> import consort
>>> harmonic_expression = consort.HarmonicExpression()
>>> print(format(harmonic_expression))
consort.tools.HarmonicExpression(
   touch_interval=pitchtools.NamedInterval('+P4'),
)
::
>>> staff = Staff("c'4 d'4 ~ d'4 e'4")
>>> logical_tie = inspect_(staff[l]).get_logical_tie()
>>> harmonic_expression(logical_tie)
>>> print(format(staff))
\new Staff {
c'4  
<
d' 
\tweak #'style #'harmonic  
g'
>4 ~ 
<
d'  
\tweak #'style #'harmonic  
g'
>4  
e'4  
}
'''

### CLASS VARIABLES ###
__slots__ = ('_touch_interval',)

### INITIALIZER ###
def __init__(
   self,
   touch_interval='P4',
):
   touch_interval = pitchtools.NamedInterval(touch_interval)
   self._touch_interval = touch_interval

### SPECIAL METHODS ###
```python
def __call__(
    self,
    logical_tie,
    pitch_range=None,
):  
    for i, leaf in enumerate(logical_tie):
        stopped_pitch = leaf.written_pitch
        touched_pitch = stopped_pitch.transpose(self.touch_interval)
        chord = scoretools.Chord(leaf)
        chord.written_pitches = [stopped_pitch, touched_pitch]
        #chord.note_heads[0].is_parenthesized = True
        #chord.note_heads[0].tweak.font_size = -4
        chord.note_heads[1].tweak.style = 'harmonic'
        self._replace(leaf, chord)

### PUBLIC PROPERTIES ###

@property
def touch_interval(self):
    return self._touch_interval
```

---

### A.23 consorttools.HashCachingObject ###

```python
# -*- encoding: utf-8 -*-
from abjad import abctools

class HashCachingObject(abctools.AbjadValueObject):

    ### CLASS VARIABLES ###

    __slots__ = ('_hash',)

    ### INITIALIZER ###

def __init__(self):
    self._hash = None

    ### SPECIAL METHODS ###

def __eq__(self, expr):
    if isinstance(expr, type(self)):
        if format(self) == format(expr):
            return True
        return False

def __hash__(self):
    if self._hash is None:
        hash_values = (type(self), format(self))
        self._hash = hash(hash_values)
    return self._hash
```
A.24 consort.tools.KeyClusterExpression

```python
# -*- encoding: utf-8 -*-
from __future__ import print_function
from abjad import attach
from abjad.tools import indicatortools
from abjad.tools import pitchtools
from abjad.tools import scoretools
from abjad.tools import selectiontools
from consort.tools.LogicalTieExpression import LogicalTieExpression

class KeyClusterExpression(LogicalTieExpression):
    r'''A key cluster expression.
    >>> import consort
    >>> key_cluster_expression = consort.KeyClusterExpression(
    ...     arpeggio_direction=Up,
    ...     include_black_keys=False,
    ...     )
    >>> print(format(key_cluster_expression))
    consort.tools.KeyClusterExpression(
        arpeggio_direction=Up,
        include_black_keys=False,
        include_white_keys=True,
        staff_space_width=5,
    )
    >>> print(format(key_cluster_expression))
    >>> staff = Staff("c'4 d'4 ~ d'4 e'4")
    >>> logical_tie = inspect_(staff[1]).get_logical_tie()
    >>> key_cluster_expression(logical_tie)
    >>> print(format(staff))
    \new Staff {
        c'4
        \arpeggioArrowUp
        \once \override Accidental.stencil = ##f
        \once \override AccidentalCautionary.stencil = ##f
        \once \override Arpeggio.X-offset = #-2
        \once \override NoteHead.stencil = \markup {\filled-box #'(-0.6 . 0.6) #'(-0.7 . 0.7) #0.25
            \center-align
            \natural
        }\markup ~
        b d' f'4 \arpeggio
        ^ \markup {
            \center-align
            \natural
        }
        \once \override Accidental.stencil = ##f
        \once \override AccidentalCautionary.stencil = ##f
        \once \override Arpeggio.X-offset = #-2
        \once \override NoteHead.stencil = \markup {\filled-box #'(-0.6 . 0.6) #'(-0.7 . 0.7) #0.25
            \center-align
            \natural
        }\markup ~
        ^ \markup {
            \center-align
            \natural
        }
```
### CLASS VARIABLES ###

```python
__slots__ = (  
    '_arpeggio_direction',  
    '_include_black_keys',  
    '_include_white_keys',  
    '_staff_space_width',  
)
```

### INITIALIZER ###

```python
def __init__(self, arpeggio_direction=None, include_black_keys=True, include_white_keys=True, staff_space_width=5,):
    assert 2 < staff_space_width and (int(staff_space_width) % 2)  
    assert include_black_keys or include_white_keys  
    assert arpeggio_direction in (Up, Down, None)  
    self._arpeggio_direction = arpeggio_direction  
    self._include_black_keys = bool(include_black_keys)  
    self._include_white_keys = bool(include_white_keys)  
    self._staff_space_width = int(staff_space_width)
```

### SPECIAL METHODS ###

```python
def __call__(self, logical_tie, pitch_range=None,):
    assert isinstance(logical_tie, selectiontools.LogicalTie), logical_tie  
    center_pitch = logical_tie[0].written_pitch  
    chord_pitches = self._get_chord_pitches(center_pitch)  
    if pitch_range is not None:
        maximum_pitch = max(chord_pitches)  
        minimum_pitch = min(chord_pitches)  
        if maximum_pitch not in pitch_range:
            interval = maximum_pitch - pitch_range.stop_pitch  
            center_pitch = center_pitch.transpose(interval)  
        chord_pitches = self._get_chord_pitches(center_pitch)  
        elif minimum_pitch not in pitch_range:
            interval = minimum_pitch - pitch_range.start_pitch  
            center_pitch = center_pitch.transpose(interval)
```
chord_pitches = self._get_chord_pitches(center_pitch)

for i, leaf in enumerate(logical_tie):
    chord = scoretools.Chord(leaf)
    chord.written_pitches = chord_pitches
    self._replace(leaf, chord)
    if i:
        key_cluster = indicatortools.KeyCluster(
            include_black_keys=self.include_black_keys,
            include_white_keys=self.include_white_keys,
            suppress_markup=True,
        )
        attach(key_cluster, chord)
    else:
        key_cluster = indicatortools.KeyCluster(
            include_black_keys=self.include_black_keys,
            include_white_keys=self.include_white_keys,
            markup_direction=Up,
        )
        attach(key_cluster, chord)
        if self.arpeggio_direction is not None:
            arpeggio = indicatortools.Arpeggio(
                direction=self.arpeggio_direction,
            )
            attach(arpeggio, chord)

### PRIVATE PROPERTIES ###

def _get_chord_pitches(self, center_pitch):
    starting_diatonic_number = \
        center_pitch.diatonic_pitch_number - (self.staff_space_width // 2)
    diatonic_numbers = [starting_diatonic_number]
    for i in range(1, (self.staff_space_width // 2) + 1):
        step = 2 * i
        diatonic_number = starting_diatonic_number + step
        diatonic_numbers.append(diatonic_number)
    chromatic_numbers = [
        (12 * (x // 7)) +
        pitchtoolsPitchClass._diatonic_pitch_class_number_to_pitch_class_number[
            x % 7]
        for x in diatonic_numbers
    ]
    chord_pitches = [pitchtools.NamedPitch(x)
        for x in chromatic_numbers]
    return chord_pitches

### PUBLIC PROPERTIES ###

@property
def arpeggio_direction(self):
    return self._arpeggio_direction

@property
def include_black_keys(self):
    return self._include_black_keys
A.25  **consort.tools.LeafExpression**

```python
# -*- encoding: utf-8 -*-
from abjad import attach
from abjad import inspect_
from abjad import mutate
from abjad import select
from abjad.tools import durationtools
from abjad.tools import scoretools
from abjad.tools import selectiontools
from consort.tools.HashCachingObject import HashCachingObject

class LeafExpression(HashCachingObject):
    
    ### CLASS VARIABLES ###
    __slots__ = (
        '_attachments',
        '_leaf',
    )

    ### INITIALIZER ###
    def __init__(self, leaf=None, attachments=None):
        HashCachingObject.__init__(self)
        prototype = scoretools.Leaf
        if leaf is not None:
            if isinstance(leaf, prototype):
                leaf = mutate(leaf).copy()
            elif issubclass(leaf, prototype):
                leaf = leaf()
            else:
                raise ValueError(leaf)
        self._leaf = leaf
        if attachments is not None:
            attachments = tuple(attachments)
        self._attachments = attachments

    ### SPECIAL METHODS ###
```
def __call__(self, expr):
    if isinstance(expr, scoretools.Leaf):
        expr = select(expr)
    assert isinstance(expr, selectiontools.Selection)
    for i, old_leaf in enumerate(expr):
        assert isinstance(old_leaf, scoretools.Leaf)
        new_leaf = self._make_new_leaf(old_leaf)

        timespan = old_leaf._timespan
        start_offset = old_leaf._start_offset
        stop_offset = old_leaf._stop_offset
        mutate(old_leaf).replace(new_leaf)
        new_leaf._timespan = timespan
        new_leaf._start_offset = start_offset
        new_leaf._stop_offset = stop_offset

        if i == 0 and self.attachments:
            for attachment in self.attachments:
                attach(attachment, new_leaf)

    ### PRIVATE METHODS ###

def _make_new_leaf(self, old_leaf):
    duration = old_leaf.written_duration
    if isinstance(self.leaf, scoretools.Note):
        new_leaf = scoretools.Note(self.leaf.written_pitch, duration)
    elif isinstance(self.leaf, scoretools.Chord):
        new_leaf = scoretools.Chord(self.leaf.written_pitches, duration)
    elif isinstance(self.leaf, scoretools.Rest):
        new_leaf = scoretools.Rest(duration)
    elif isinstance(self.leaf, scoretools.Skip):
        new_leaf = scoretools.Skip(duration)
    prototype = durationtools.Multiplier
    if inspect_(old_leaf).has_indicator(prototype):
        multiplier = inspect_(old_leaf).get_indicator(prototype)
        attach(multiplier, new_leaf)
    return new_leaf

    ### PUBLIC PROPERTIES ###

@property
def attachments(self):
    return self._attachments

@property
def leaf(self):
    return self._leaf

A.26 consort.tools.LogicalTieExpression

# -*- encoding: utf-8 -*-
import abc
from abjad import attach
detach
from abjad import inspect_
from abjad import mutate
from abjad.tools import abctools
from abjad.tools import scoretools

class LogicalTieExpression(abctools.AbjadValueObject):
    ### CLASS VARIABLES ###
    __slots__ = ()
    ### SPECIAL METHODS ###
    @abc.abstractmethod
    def __call__(self, logical_tie, pitch_range=None):
        raise NotImplementedError
    ### PRIVATE METHODS ###
    def _replace(self, old_leaf, new_leaf):
        grace_containers = inspect_(old_leaf).get_grace_containers('after')
        if grace_containers:
            old_grace_container = grace_containers[0]
            grace_notes = old_grace_container.select_leaves()
            detach(scoretools.GraceContainer, old_leaf)
            indicators = inspect_(old_leaf).get_indicators()
            for indicator in indicators:
                detach(indicator, old_leaf)
            timespan = old_leaf._timespan
            start_offset = old_leaf._start_offset
            stop_offset = old_leaf._stop_offset
            mutate(old_leaf).replace(new_leaf)
            new_leaf._timespan = timespan
            new_leaf._start_offset = start_offset
            new_leaf._stop_offset = stop_offset
            if grace_containers:
                new_grace_container = scoretools.GraceContainer(
                    grace_notes,
                    kind='after',
                )
                attach(new_grace_container, new_leaf)
                for indicator in indicators:
                    attach(indicator, new_leaf)

A.27  consort.tools.MusicSetting
from abjad.tools import abctools
from abjad.tools import systemtools
from abjad.tools import timespantools

class MusicSetting(abctools.AbjadValueObject):
    r'''A music setting.

    ::
    >>> import consort
    >>> red_setting = consort.MusicSetting(
    ...     timespan_maker=consort.TaleaTimespanMaker(
    ...         initial_silence_talea=rhythmmakertools.Talea(
    ...             counts=(0, 4),
    ...             denominator=16,
    ...         ),
    ...         playing_talea=rhythmmakertools.Talea(
    ...             counts=(4, 8, 4),
    ...             denominator=16,
    ...         ),
    ...     ),
    ...     viola_rh=consort.tools.MusicSpecifier(),
    ...     violin_1_rh=consort.tools.MusicSpecifier(),
    ...     violin_2_rh=consort.tools.MusicSpecifier(),
    ...     )
    >>> print(format(red_setting))
    consort.tools.MusicSetting(
        timespan_maker=consort.tools.TaleaTimespanMaker(
            initial_silence_talea=rhythmmakertools.Talea(
                counts=(0, 4),
                denominator=16,
            ),
            playing_talea=rhythmmakertools.Talea(
                counts=(4, 8, 4),
                denominator=16,
            ),
            playing_groupings=(1,),
            repeat=True,
            silence_talea=rhythmmakertools.Talea(
                counts=(4,),
                denominator=16,
            ),
            step_anchor=Right,
            synchronize_groupings=False,
            synchronize_step=False,
        ),
        viola_rh=consort.tools.MusicSpecifier(),
        violin_1_rh=consort.tools.MusicSpecifier(),
        violin_2_rh=consort.tools.MusicSpecifier(),
    )
    ::
>>> layer = 1

>>> segment_timespan = timespantools.Timespan(1, 2)

>>> from abjad.tools import templatetools

>>> score_template = consort.StringQuartetScoreTemplate()

>>> timespan_inventory = red_setting(
...     layer=layer,
...     score_template=score_template,
...     segment_timespan=segment_timespan,
... )

>>> print(format(timespan_inventory))

```
timespantools.TimespanInventory([  
consort.tools.PerformedTimespan(  
    start_offset=durationtools.Offset(1, 1),  
    stop_offset=durationtools.Offset(5, 4),  
    layer=1,  
    music_specifier=consort.tools.MusicSpecifier(),  
    voice_name='Violin 1 Bowing Voice',  
  ),  
consort.tools.PerformedTimespan(  
    start_offset=durationtools.Offset(1, 1),  
    stop_offset=durationtools.Offset(3, 2),  
    layer=1,  
    music_specifier=consort.tools.MusicSpecifier(),  
    voice_name='Viola Bowing Voice',  
  ),  
consort.tools.PerformedTimespan(  
    start_offset=durationtools.Offset(5, 4),  
    stop_offset=durationtools.Offset(3, 2),  
    layer=1,  
    music_specifier=consort.tools.MusicSpecifier(),  
    voice_name='Violin 2 Bowing Voice',  
  ),  
consort.tools.PerformedTimespan(  
    start_offset=durationtools.Offset(3, 2),  
    stop_offset=durationtools.Offset(2, 1),  
    layer=1,  
    music_specifier=consort.tools.MusicSpecifier(),  
    voice_name='Violin 1 Bowing Voice',  
  ),  
consort.tools.PerformedTimespan(  
    start_offset=durationtools.Offset(7, 4),  
    stop_offset=durationtools.Offset(2, 1),  
    layer=1,  
    music_specifier=consort.tools.MusicSpecifier(),  
    voice_name='Viola Bowing Voice',  
  ),  
consort.tools.PerformedTimespan(  
    start_offset=durationtools.Offset(7, 4),  
    stop_offset=durationtools.Offset(2, 1),  
    layer=1,  
```
>>> red_setting = new(
    ... red_setting,
    ... silenced_contexts=[
    ... 'viola_lh',
    ... 'cello',
    ... ],
    ... )

>>> timespan_inventory = red_setting(
    ... layer=layer,
    ... score_template=score_template,
    ... segment_timespan=segment_timespan,
    ... )

>>> print(format(timespan_inventory))
timespantools.TimespanInventory([
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(1, 1),
        stop_offset=durationtools.Offset(5, 4),
        layer=1,
        music_specifier=consort.tools.MusicSpecifier(),
        voice_name='Violin 1 Bowing Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(1, 1),
        stop_offset=durationtools.Offset(3, 2),
        layer=1,
        music_specifier=consort.tools.MusicSpecifier(),
        voice_name='Viola Bowing Voice',
    ),
    consort.tools.SilentTimespan(
        start_offset=durationtools.Offset(1, 1),
        stop_offset=durationtools.Offset(2, 1),
        layer=1,
        voice_name='Cello Bowing Voice',
    ),
    consort.tools.SilentTimespan(
        start_offset=durationtools.Offset(1, 1),
        stop_offset=durationtools.Offset(2, 1),
        layer=1,
        voice_name='Viola Fingering Voice',
    ),
    consort.tools.SilentTimespan(
        start_offset=durationtools.Offset(1, 1),
        stop_offset=durationtools.Offset(2, 1),
        layer=1,
        voice_name='Cello Fingering Voice',
    ),
    consort.tools.SilentTimespan(
        start_offset=durationtools.Offset(1, 1),
        stop_offset=durationtools.Offset(2, 1),
        layer=1,
        voice_name='Viola Fingering Voice',
    ),
]
```
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(5, 4),
        stop_offset=durationtools.Offset(3, 2),
        layer=1,
        music_specifier=consort.tools.MusicSpecifier(),
        voice_name='Violin 2 Bowing Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(3, 2),
        stop_offset=durationtools.Offset(2, 1),
        layer=1,
        music_specifier=consort.tools.MusicSpecifier(),
        voice_name='Violin 1 Bowing Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(7, 4),
        stop_offset=durationtools.Offset(2, 1),
        layer=1,
        music_specifier=consort.tools.MusicSpecifier(),
        voice_name='Viola Bowing Voice',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(7, 4),
        stop_offset=durationtools.Offset(2, 1),
        layer=1,
        music_specifier=consort.tools.MusicSpecifier(),
        voice_name='Violin 2 Bowing Voice',
    ),
)
```

### CLASS VARIABLES ###

```python
__slots__ = ('_music_specifiers',
             '_silenced_contexts',
             '_timespan_identifier',
             '_timespan_maker',
)
```

### INITIALIZER ###

```python
def __init__(
    self,
    timespan_identifier=None,
    timespan_maker=None,
    silenced_contexts=None,
    **music_specifiers
    ):
    import consort
    prototype = (  
```
consort.CompositeMusicSpecifier,
consort.MusicSpecifier,
consort.MusicSpecifierSequence,
str, # for demonstration purposes only
type(None),
}

for abbreviation, music_specifier in sorted(music_specifiers.items()):
    if isinstance(music_specifier, prototype):
        continue
    elif isinstance(music_specifier, collections.Sequence) and all(isinstance(x, prototype) for x in music_specifier):
        music_specifier = consort.MusicSpecifierSequence(
            music_specifiers=music_specifier,
        )
        music_specifiers[abbreviation] = music_specifier
    else:
        raise ValueError(music_specifier)
    self._music_specifiers = music_specifiers

if silenced_contexts is not None:
    silenced_contexts = (str(_).encode('utf-8') for _ in silenced_contexts)
    silenced_contexts = set(silenced_contexts)
    self._silenced_contexts = silenced_contexts

if timespan_identifier is not None:
    prototype = (timespantools.Timespan,
                 timespantools.TimespanInventory,
                 consort.RatioPartsExpression,
                 )
    if not isinstance(timespan_identifier, prototype):
        timespan_identifier = \ 
        consort.RatioPartsExpression.from_sequence(timespan_identifier)

    assert isinstance(timespan_identifier, prototype)
    self._timespan_identifier = timespan_identifier

    if timespan_maker is not None:
        assert isinstance(timespan_maker,
                         consort.TimespanMaker), \ 
        timespan_maker
    else:
        timespan_maker = consort.FloodedTimespanMaker()
    self._timespan_maker = timespan_maker

    if score is None:

### SPECIAL METHODS ###

def __call__(
    self,
    layer=None,
    score=None,
    score_template=None,
    segment_timespan=None,
    timespan_inventory=None,
    timespan_quantization=None,
):
score = score_template()
if timespan_inventory is None:
    timespan_inventory = timespantools.TimespanInventory()
if not self.music_specifiers:
    return timespan_inventory
music_specifiers = self.resolve_music_specifiers(
    score_template, score=score,
)
silenced_context_names = self.resolve_silenced_contexts(
    score_template, score=score,
)
target_timespans = self.resolve_target_timespans(
    segment_timespan, timespan_quantization,
)
for i, target_timespan in enumerate(target_timespans):
    timespan_maker = self.timespan_maker.rotate(i)
    timespan_inventory = timespan_maker(
        layer=layer,
        music_specifiers=music_specifiers,
        silenced_context_names=silenced_context_names,
        target_timespan=target_timespan,
        timespan_inventory=timespan_inventory,
    )
return timespan_inventory

def __getattr__(self, item):
    if item in self.music_specifiers:
        return self.music_specifiers[item]
    return object.__getattribute__(self, item)

### PRIVATE PROPERTIES ###
@property
def _storage_format_specification(self):
    manager = systemtools.StorageFormatManager
    keyword_argument_names = manager.get_keyword_argument_names(self)
    keyword_argument_names = list(keyword_argument_names)
    keyword_argument_names.extend(sorted(self.music_specifiers))
    return systemtools.StorageFormatSpecification(
        self,
        keyword_argument_names=keyword_argument_names
    )

### PUBLIC METHODS ###
def resolve_music_specifiers(self, score_template, score=None,):
    import consort
assert score_template is not None
if score is None:
    score = score_template()
all_abbreviations = score_template.context_name_abbreviations
prototype = (consort.CompositeMusicSpecifier, consort.MusicSpecifierSequence,)
triples = []
for abbreviation, music_specifier in self.music_specifiers.items():
    if not isinstance(music_specifier, prototype):
        music_specifier = consort.MusicSpecifierSequence(
            music_specifiers=music_specifier,
        )
    context_name = all_abbreviations[abbreviation]
    context = score[context_name]
    context_index = inspect_(context).get_parentage().score_index
    context_name = context.name
    if isinstance(music_specifier, consort.CompositeMusicSpecifier):
        composite_pairs = score_template.composite_context_pairs
        one, two = composite_pairs[abbreviation]
        primary_voice_name = all_abbreviations[one]
        secondary_voice_name = all_abbreviations[two]
        music_specifier = new(
            music_specifier,
            primary_voice_name=primary_voice_name,
            secondary_voice_name=secondary_voice_name,
        )
        triple = (
            context_index,
            context_name,
            music_specifier,
        )
        triples.append(triple)
triples.sort(key=lambda x: x[0])
music_specifiers = collections.OrderedDict()
for context_index, context_name, music_specifier in triples:
    music_specifiers[context_name] = music_specifier
return music_specifiers

def resolve_silenced_contexts(
    self,
    score_template,
    score=None,
):
    assert score_template is not None
    if score is None:
        score = score_template()
    all_abbreviations = score_template.context_name_abbreviations
    composite_pairs = getattr(score_template, 'composite_context_pairs',)
    ()

537
silenced_context_names = set()
silenced_contexts = self.silenced_contexts or ()

for abbreviation in silenced_contexts:
    if abbreviation in composite_pairs:
        one, two = composite_pairs[abbreviation]
        primary_voice_name = all_abbreviations[one]
        secondary_voice_name = all_abbreviations[two]
        silenced_context_names.add(primary_voice_name)
        silenced_context_names.add(secondary_voice_name)
    elif abbreviation in all_abbreviations:
        context_name = all_abbreviations[abbreviation]
        silenced_context_names.add(context_name)
    else:
        message = 'Unresolvable context abbreviation: {}'
        message = message.format(abbreviation)
        raise Exception(message)

return silenced_context_names

def resolve_target_timespans(
    self,
    segment_timespan,
    timespan_quantization=None,
):
    import consort
    assert isinstance(segment_timespan, timespantools.Timespan)
timespan_identifier = self.timespan_identifier

    if timespan_identifier is None:
        target_timespans = timespantools.TimespanInventory([segment_timespan,])
    elif isinstance(self.timespan_identifier, timespantools.Timespan):
        if timespan_identifier.stop_offset == Infinity:
            timespan_identifier = new(
                timespan_identifier,
                stop_offset=segment_timespan.stop_offset,
            )
        segment_timespan = timespantools.Timespan(start_offset=0)
        target_timespans = segment_timespan & timespan_identifier
    else:
        if isinstance(timespan_identifier, consort.RatioPartsExpression):
            mask_timespans = timespan_identifier(segment_timespan)
        else:
            mask_timespans = timespan_identifier
        target_timespans = timespantools.TimespanInventory()
        for mask_timespan in mask_timespans:
            available_timespans = segment_timespan & mask_timespan
            target_timespans.extend(available_timespans)

    if timespan_quantization is not None:
        target_timespans.round_offsets(
            timespan_quantization,
            must_be_well_formed=True,
        )

    return target_timespans
### PUBLIC PROPERTIES ###

```python
@property
def music_specifiers(self):
    return self._music_specifiers

@property
def silenced_contexts(self):
    return self._silenced_contexts

@property
def timespan_identifier(self):
    return self._timespan_identifier

@property
def timespan_maker(self):
    return self._timespan_maker
```

---

A.28  **consort.tools.MusicSpecifier**

```python
from consort.tools.HashCachingObject import HashCachingObject

class MusicSpecifier(HashCachingObject):
    r'''A music specifier.''

    >>> import consort
    >>> music_specifier = consort.MusicSpecifier()
    >>> print(format(music_specifier))
    consort.tools.MusicSpecifier()

    .. container:: example

        MusicSpecifier can accept CompositeRhythmMakers in their ‘rhythm_maker’
        slot:

        ::

            >>> music_specifier = consort.MusicSpecifier(
            ...     rhythm_maker=consort.CompositeRhythmMaker(),
```
... )

```

### CLASS VARIABLES ###

__slots__ = ('_attachment_handler',
             '_color',
             '_grace_handler',
             '_instrument',
             '_labels',
             '_minimum_phrase_duration',
             '_pitch_handler',
             '_rhythm_maker',
             '_seed',
)

### INITIALIZER ###

def __init__(self,
              attachment_handler=None,
              color=None,
              grace_handler=None,
              instrument=None,
              labels=None,
              minimum_phrase_duration=None,
              pitch_handler=None,
              rhythm_maker=None,
              seed=None,
):  
  import consort  
  HashCachingObject.__init__(self)  
  if attachment_handler is not None:  
    assert isinstance(attachment_handler, consort.AttachmentHandler)  
  self._attachment_handler = attachment_handler  
  if color is not None:  
    color = str(color)  
  self._color = color  
  if grace_handler is not None:  
    assert isinstance(grace_handler, consort.GraceHandler)  
  self._grace_handler = grace_handler  
  if instrument is not None:  
    assert isinstance(instrument, instrumenttools.Instrument)  
  self._instrument = instrument  
  if labels is not None:  
    if isinstance(labels, str):
      labels = (labels,)  
      labels = tuple(str(_) for _ in labels)  
  self._labels = labels  
  if minimum_phrase_duration is not None:  
    minimum_phrase_duration = \
    durationtools.Duration(minimum_phrase_duration)
```python
assert 0 <= minimum_phrase_duration

self._minimum_phrase_duration = minimum_phrase_duration

if pitch_handler is not None:
    assert isinstance(pitch_handler, consort.PitchHandler)

self._pitch_handler = pitch_handler

if rhythm_maker is not None:
    prototype = (  # noqa: E501
        rhythmmakertools.RhythmMaker,
        consort.CompositeRhythmMaker,
    )
    assert isinstance(rhythm_maker, prototype)

self._rhythm_maker = rhythm_maker

if seed is not None:
    seed = int(seed)

self._seed = seed

### SPECIAL METHODS ###

def __illustrate__(
    self,
    annotate=False,
    verbose=True,
    package_name=None,
    **kwargs
):

    r'''Illustrates music specifier.

    ::

        >>> piano_glissando_music_specifier = consort.MusicSpecifier(
        ...     attachment_handler=consort.AttachmentHandler(
        ...         glissando=spannertools.Glissando(),
        ...     ),
        ...     color=None,
        ...     labels=[],
        ...     pitch_handler=consort.AbsolutePitchHandler(
        ...         pitch_specifier="c' f' c'' f'' c''' c' c'''",
        ...     ),
        ...     rhythm_maker=consort.CompositeRhythmMaker(
        ...         last=rhythmmakertools.IncisedRhythmMaker(
        ...             inciseSpecifier=rhythmmakertools.InciseSpecifier(
        ...                 prefix_counts=[0],
        ...                 suffix_talea=[1],
        ...                 suffix_counts=[1],
        ...                 talea_denominator=16,
        ...             ),
        ...         ),
        ...     ),
        ...     duration_spellingSpecifier=rhythmmakertools.DurationSpellingSpecifier(
        ...         decrease_durations_monotonically=True,
        ...         forbidden_written_duration=(1, 4),
        ...         forbid_meter_rewriting=True,
        ...     ),
    ...
```
... ),
... ),
... )

>>> illustration = piano_glissando_music_specifier.__illustrate__(
... annotate=True,
... verbose=False,
... )

Returns LilyPond file.

import consort

score_template = consort.StringQuartetScoreTemplate(
    split=False,
    without_instruments=True,
)

segment_maker = consort.SegmentMaker(
    desired_duration_in_seconds=10,
    discard_final_silence=True,
    #annotate_phrasing=True,
    annotate_phrasing=annotate,
    permitted_time_signatures=[
        (3, 8),
        (4, 8),
        (5, 8),
        (6, 8),
        (7, 8),
    ],
    score_template=score_template,
    tempo=indicatortools.Tempo((1, 4), 72),
    timespan_quantization=(1, 8),
)

timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=rhythmmakertools.Talea(
        counts=[0, 2, 1],
        denominator=8,
    ),
    playing_talea=rhythmmakertools.Talea(
        counts=[3, 2, 1, 4, 5, 3, 1, 2],
        denominator=8,
    ),
    playing_groupings=[2, 1, 2, 3, 1, 3, 4, 1, 2, 3],
    repeat=True,
    silence_talea=rhythmmakertools.Talea(
        counts=[1, 2, 3, 1, 2],
        denominator=8,
    ),
    step_anchor=Right,
    synchronize_groupings=False,
    synchronize_step=False,
    timespan_specifier=consort.TimespanSpecifier(
        minimum_duration=durationtools.Duration(1, 8),
    ),
)

segment_maker.add_setting
timespan_maker=timespan_maker,
    violin_1=self,
    violin_2=self,
    viola=self,
    cello=self,
)
segment_metadata = collections.OrderedDict(
    segment_count=1,
    segment_number=1,
)
lilypond_file, segment_metadata = segment_maker(
    segment_metadata=segment_metadata,
    verbose=verbose,
)
consort_stylesheets_path = os.path.join(
    consort.__path__[0],
    'stylesheets',
    'stylesheet.ily',
)
consort_stylesheets_path = os.path.abspath(consort_stylesheets_path)
lilypond_file.file_initial_user_includes[:] = [consort_stylesheets_path]
lilypond_file.use_relative_includes = False
if package_name is not None:
    header = lilypondfiletools.Block('header')
    header.title = stringtools.to_space_delimited_lowercase(
        package_name).title()
    header.tagline = markuptools.Markup('””')
lilypond_file.items.insert(0, header)
return lilypond_file

### PUBLIC METHODS ###

def rotate(self, rotation):
    seed = self.seed or 0
    seed = seed + rotation
    return new(self, seed=seed)

def transpose(self, expr):
    r'''Transposes music specifier.
    ::

    >>> music_specifier = consort.MusicSpecifier(
    ...     pitch_handler=consort.AbsolutePitchHandler(
    ...         pitchSpecifier = consort.PitchSpecifier(
    ...             pitch_segments=(
    ...                 "c' e' g'",
    ...                 "fs' gs'",
    ...                 "b",
    ...             ),
    ...             ratio=(1, 2, 3),
    ...         ),
    ...     ),
    ... )
    ... )
    ... )
>>> transposed_musicSpecifier = musicSpecifier.transpose('-M2')
>>> print(format(transposed_musicSpecifier))

consort.tools.MusicSpecifier(
    pitch_handler=consort.tools.AbsolutePitchHandler(
        pitch_specifier=consort.tools.PitchSpecifier(
            pitch_segments=(
                pitchtools.PitchSegment(
                    pitchtools.NamedPitch('bf'),
                    pitchtools.NamedPitch('d'''),
                    pitchtools.NamedPitch('f'''),
                    item_class=pitchtools.NamedPitch,
                    ),
                pitchtools.PitchSegment(
                    pitchtools.NamedPitch('e'''),
                    pitchtools.NamedPitch('fs'''),
                    item_class=pitchtools.NamedPitch,
                    ),
                pitchtools.PitchSegment(
                    pitchtools.NamedPitch('a'),
                    item_class=pitchtools.NamedPitch,
                    ),
                ),
            ratio=mathtools.Ratio((1, 2, 3)),
        ),
    ),
)

Returns new music specifier.

if isinstance(expr, str):
    try:
        pitch = pitchtools.NamedPitch(expr)
        expr = pitchtools.NamedPitch('C4') - pitch
    except:
        expr = pitchtools.NamedInterval(expr)
    pitch_handler = self.pitch_handler
    if pitch_handler is not None:
        pitch_handler = pitch_handler.transpose(expr)
    return new(
        self,
        pitch_handler=pitch_handler,
    )

### PUBLIC PROPERTIES ###

@property
def attachment_handler(self):
    return self._attachment_handler
A.29 consort.tools.MusicSpecifierSequence

```python
# -*- encoding: utf-8 -*-

import collections

from abjad import new
from abjad.tools import abctools
from abjad.tools import datastructuretools
from abjad.tools import mathtools
from abjad.tools import rhythmtools
from abjad.tools import sequencetools
from abjad.tools import timespantools

class MusicSpecifierSequence(abctools.AbjadValueObject):
    r'''A music specifier sequence.

    ::

        >>> import consort
        >>> sequence_a = consort.MusicSpecifierSequence(
        ...     music_specifiers='music',
        ...     )
```
>>> print(format(sequence_a))
consort.tools.MusicSpecifierSequence(
    music_specifiers=datastructuretools.CyclicTuple(['music']),
)

::

>>> sequence_b = consort.MusicSpecifierSequence(
    application_rate='phrase',
    music_specifiers=['one', 'two', 'three'],
)
>>> print(format(sequence_b))
consort.tools.MusicSpecifierSequence(
    application_rate='phrase',
    music_specifiers=datastructuretools.CyclicTuple(['one', 'two', 'three']),
)

### CLASS VARIABLES ###

__slots__ = ('_application_rate', '_music_specifiers',)

### INITIALIZER ###

def __init__(
    self,
    application_rate=None,
    music_specifiers=None,
):
    if application_rate is not None:
        application_rate = application_rate or 'phrase'
        assert application_rate in ('division', 'phrase')
    if music_specifiers is not None:
        if not isinstance(music_specifiers, collections.Sequence) or \
            not isinstance(music_specifiers, str):
            music_specifiers = [music_specifiers]
        music_specifiers = tuple(music_specifiers)
        music_specifiers = datastructuretools.CyclicTuple(music_specifiers)
        assert len(music_specifiers)
        self._application_rate = application_rate
        self._music_specifiers = music_specifiers

### SPECIAL METHODS ###

def __call__(

546
    self,
    durations=None,
    layer=None,
    output_masks=None,
    padding=None,
    seed=None,
    start_offset=None,
    timespanSpecifier=None,
    voice_name=None,
):  
        import consort
        timespans = timespantools.TimespanInventory()
        timespanSpecifier = timespanSpecifier or 
            consort.TimespanSpecifier()
        seed = seed or 0
        durations = [d for d in durations if d]
        offsets = mathtools.cumulative_sums(durations, start_offset)
        if not offsets:
            return timespans
        offset_pair_count = len(offsets) - 1
        output_mask_prototype = (
            type(None),
            rhythmmakertools.SustainMask,
        )
        iterator = sequencetools.iterate_sequence_nwise(offsets)
        for i, offset_pair in enumerate(iterator):
            start_offset, stop_offset = offset_pair
            musicSpecifier = self[seed]
            timespan = consort.PerformedTimespan(
                forbid_fusing=timespanSpecifier.forbid_fusing,
                forbid_splitting=timespanSpecifier.forbid_splitting,
                layer=layer,
                minimum_duration=timespanSpecifier.minimum_duration,
                musicSpecifier=musicSpecifier,
                start_offset=start_offset,
                stop_offset=stop_offset,
                voice_name=voice_name,
            )
            if not output_masks:
                timespans.append(timespan)
            else:
                output_mask = output_masks.get_matching_pattern(
                    i, offset_pair_count, rotation=seed)
                if isinstance(output_mask, output_mask_prototype):
                    timespans.append(timespan)
            if self.application_rate == 'division':
                seed += 1
        if padding:
            silent_timespans = timespantools.TimespanInventory()
            for shard in timespans.partition(True):
                silent_timespan_one = consort.SilentTimespan(
                    layer=layer,
A.30 \texttt{consort.tools.OctavationExpression}
from abjad.tools import scoretools
from abjad.tools import spannertools

class OctavationExpression(abctools.AbjadValueObject):
    r'''An octavation expression.
    
    ### CLASS VARIABLES ###
    __slots__ = ()
    
    ### INITIALIZER ###
    def __init__(self):
        pass
    
    ### SPECIAL METHODS ###
    def __call__(self, music, name=None):
        leaves = music.select_leaves()
        weights = []
        weighted_pitches = []
        for leaf in leaves:
            weight = float(inspect_(leaf).get_duration())
            if isinstance(leaf, scoretools.Note):
                pitch = float(leaf.written_pitch)
                weighted_pitch = pitch * weight
                weights.append(weight)
                weighted_pitches.append(weighted_pitch)
            elif isinstance(leaf, scoretools.Chord):
                for pitch in leaf.written_pitches:
                    pitch = float(pitch)
                    weighted_pitch = pitch * weight
                    weighted_pitches.append(weighted_pitch)
                    weights.append(weight)
        sum_of_weights = sum(weights)
        sum_of_weighted_pitches = sum(weighted_pitches)
        weighted_average = sum_of_weighted_pitches / sum_of_weights
        clef = inspect_(leaves[0]).get_effective(indicatortools.Clef)
        octavation_spanner = None
        if clef == indicatortools.Clef('treble'):
            if int(pitchtools.NamedPitch('C6')) <= int(weighted_average):
                octavation_spanner = spannertools.OctavationSpanner()
        elif clef == indicatortools.Clef('bass'):
            pass
        if octavation_spanner is not None:
            attach(octavation_spanner, music)
```
from abjad.tools import markuptools
from abjad.tools import mathtools
from abjad.tools import timespantools

class PerformedTimespan(timespantools.Timespan):
    r'''
    A Consort timespan.
    ::
    >>> import consort
    >>> timespan = consort.PerformedTimespan()
    >>> print(format(timespan))
    consort.tools.PerformedTimespan(
        start_offset=NegativeInfinity,
        stop_offset=Infinity,
    )
    '''

    ### CLASS VARIABLES ###

    __slots__ =
    (_forbid_fusing,
     _forbid_splitting,
     _divisions,
     _layer,
     _minimum_duration,
     _music,
     _music_specifier,
     _original_start_offset,
     _original_stop_offset,
     _voice_name,
    )

    ### INITIALIZER ###

    def __init__(
        self,
        start_offset=mathtools.NegativeInfinity(),
        stop_offset=mathtools.Infinity(),
        divisions=None,
        forbid_fusing=None,
        forbid_splitting=None,
        layer=None,
        minimum_duration=None,
        music=None,
        music_specifier=None,
        original_start_offset=None,
        original_stop_offset=None,
        voice_name=None,
    ):
        timespantools.Timespan.__init__(
            self,
        )
```
start_offset=start_offset,
stop_offset=stop_offset,
)

if divisions is not None:
divisions = tuple(durationtools.Duration(_) for _ in divisions)
assert sum(divisions) == self.duration
self._divisions = divisions
if forbid_fusing is not None:
    forbid_fusing = bool(forbid_fusing)
self._forbid_fusing = forbid_fusing
if forbid_splitting is not None:
    forbid_splitting = bool(forbid_splitting)
self._forbid_splitting = forbid_splitting
if layer is not None:
    layer = int(layer)
self._layer = layer
if minimum_duration is not None:
    minimum_duration = durationtools.Duration(minimum_duration)
if music is not None:
    self._music = music
if music_specifier is not None:
    self._music_specifier = music_specifier
if original_start_offset is not None:
    original_start_offset = durationtools.Offset(original_start_offset)
else:
    original_start_offset = self.start_offset
if original_stop_offset is not None:
    original_stop_offset = durationtools.Offset(original_stop_offset)
else:
    original_stop_offset = self.stop_offset
self._original_stop_offset = original_stop_offset
self._voice_name = voice_name

### PRIVATE PROPERTIES ###

@property
def _storage_format_specification(self):
    from abjad.tools import systemtools
    manager = systemtools.StorageFormatManager
    keyword_argument_names = list(manager.get_keyword_argument_names(self))
    if self._original_start_offset == self.start_offset:
        keyword_argument_names.remove('original_start_offset')
    if self._original_stop_offset == self.stop_offset:
        keyword_argument_names.remove('original_stop_offset')
    return systemtools.StorageFormatSpecification(
        self,
        keyword_argument_names=keyword_argument_names,
    )
### SPECIAL METHODS ###

```python
def __lt__(self, expr):
    if timespanTools.Timespan.__lt__(self, expr):
        return True
    if not timespanTools.Timespan.__gt__(self, expr):
        if hasattr(expr, 'voice_name'):
            return self.voice_name < expr.voice_name
        return False
```

### PRIVATE METHODS ###

```python
def _as_postscript(self, postscript_x_offset, postscript_y_offset, postscript_scale):
    start = (float(self.start_offset) * postscript_scale)
    start -= postscript_x_offset
    stop = (float(self.stop_offset) * postscript_scale)
    stop -= postscript_x_offset
    ps = markuptools.Postscript()
    ps = ps.moveto(start, postscript_y_offset)
    ps = ps.lineto(stop, postscript_y_offset)
    ps = ps.stroke()
    ps = ps.moveto(start, postscript_y_offset + 0.75)
    ps = ps.lineto(start, postscript_y_offset - 0.75)
    ps = ps.stroke()
    ps = ps.moveto(stop, postscript_y_offset + 0.75)
    ps = ps.lineto(stop, postscript_y_offset - 0.75)
    ps = ps.stroke()
    if self.layer is not None:
        ps = ps.moveto(start, postscript_y_offset)
        ps = ps.rmoveto(0.25, 0.5)
        #ps = ps.scale(0.8, 0.8)
        ps = ps.show(str(self.layer))
        #ps = ps.scale(1.25, 1.25)
        return ps
```

### PUBLIC PROPERTIES ###

```python
@property
def divisions(self):
    return self._divisions

@property
def forbid_fusing(self):
    return self._forbid_fusing

@property
def forbid_splitting(self):
    return self._forbid_splitting
```
@property
def is_left_broken(self):
    if self.original_start_offset is not None:
        if self.original_start_offset != self.start_offset:
            return True
    return False

@property
def is_right_broken(self):
    if self.original_stop_offset is not None:
        if self.original_stop_offset != self.stop_offset:
            return True
    return False

@property
def layer(self):
    return self._layer

@property
def minimum_duration(self):
    return self._minimum_duration

@property
def music(self):
    return self._music

@property
def music_specifier(self):
    return self._music_specifier

@property
def original_start_offset(self):
    return self._original_start_offset

@property
def original_stop_offset(self):
    return self._original_stop_offset

@property
def voice_name(self):
    return self._voice_name

A.32  consort.tools.PhrasedSelectorCallback

# -*- encoding: utf-8 -*-
from abjad.tools import abctools
from abjad.tools import durationtools
from abjad.tools import selectiontools

class PhrasedSelectorCallback(abctools.AbjadValueObject):
    ### CLASS VARIABLES ###
__slots__ = ()

### SPECIAL METHODS ###

def __call__(self, expr):
    assert isinstance(expr, tuple), repr(tuple)
    result = []
    for subexpr in expr:
        subresult = []
        for division in subexpr[:-1]:
            leaf = division.select_leaves()[0]
            selection = selectiontools.Selection(leaf)
            subresult.append(selection)
        leaves = subexpr[-1].select_leaves()
        if leaves.get_duration() <= durationtools.Duration(1, 8):
            subresult.append(leaves[-1])
        elif 1 == len(leaves):
            subresult.append(leaves[0])
        else:
            subresult.append(leaves[0])
            subresult.append(leaves[1])
        subresult = selectiontools.Selection(subresult)
        result.append(subresult)
    result = tuple(result)
    return result

A.33 consort.tools.PitchClassPitchHandler

```
>>> import consort
>>> pitch_handler = consort.PitchClassPitchHandler(
...     pitch_specifier="c' d' e' f'",
... )
>>> print(format(pitch_handler))
consort.tools.PitchClassPitchHandler(
    pitch_specifier=consort.tools.PitchSpecifier(
        pitch_segments=(
            pitchtools.NamedPitch("c'"),
            pitchtools.NamedPitch("d'"),
            pitchtools.NamedPitch("e'"),
```
```python
pitchtools.NamedPitch("f"),
),
item_class=pitchtools.NamedPitch,
),
)
ratio=mathtools.Ratio((1,)),
),
)
'''

### CLASS VARIABLES ###

__slots__ = ('_leap_constraint',
',_octavations',
',_pitch_range',
',_registerSpecifier',
',_registerSpread',
)

_default_octavations = datastructuretools.CyclicTuple([4, 2, 1, 0, 5, 6, 7, 3,
0, 5, 3, 1, 7, 4, 2, 6,
2, 1, 5, 3, 4, 0, 7, 6,
1, 2, 4, 0, 5, 7, 6, 3,
7, 0, 3, 1, 5, 4, 6, 2,
6, 1, 2, 0, 7, 5, 3, 4,
3, 0, 4, 7, 2, 5, 6, 1,
2, 3, 4, 7, 5, 1, 0, 6,
])

### INITIALIZER ###

def __init__(self,
  deviations=None,
  forbid_repetitions=None,
  grace_expressions=None,
  leap_constraint=None,
  logical_tie_expressions=None,
  octavations=None,
  pitch_application_rate=None,
  pitch_range=None,
  registerSpecifier=None,
  registerSpread=None,
  pitchOperationSpecifier=None,
  pitchSpecifier=None,
  pitchesAreNonsemantic=None,
):
  PitchHandler.__init__(self,
    deviations=deviations,
    forbid_repetitions=forbid_repetitions,
```

grace_expressions=grace_expressions,
logical_tie_expressions=logical_tie_expressions,
pitch_application_rate=pitch_application_rate,
pitch_operationSpecifier=pitch_operationSpecifier,
pitchSpecifier=pitchSpecifier,
pitches_are_nonsemantic=pitches_are_nonsemantic,
)
self._initialize_leap_constraint(leap_constraint)
self._initialize_octavations(octavations)
self._initialize_pitch_range(pitch_range)
self._initialize_registerSpecifier(registerSpecifier)
self._initialize_register_spread(register_spread)

### SPECIAL METHODS ###

def __call__(
    self,
    attack_point_signature,
    logical_tie,
    musicSpecifier,
    pitch_choices,
    previous_pitch,
    seed_session,
)
    previous_pitch_class = pitchtools.NamedPitchClass(previous_pitch)
    instrument = self._get_instrument(logical_tie, musicSpecifier)
    pitch_range = self._get_pitch_range(
        instrument,
        logical_tie,
    )
    registration = self._get_registration(
        attack_point_signature,
        logical_tie,
        seed_session.current_timewise_phrase_seed,
    )
    pitch_class = self._get_pitch_class(
        attack_point_signature,
        pitch_choices,
        previous_pitch_class,
        seed_session.current_phrased_voicewise_logical_tie_seed,
    )
    pitch = self._get_pitch(
        pitch_class,
        registration,
        seed_session.current_phrased_voicewise_logical_tie_seed,
    )
    pitch = self._constrain_interval(  
        pitch,
        previous_pitch,
    )
    pitch = self._apply_deviation(  
        pitch,
        seed_session.current_unphrased_voicewise_logical_tie_seed,
    )
pitch_range = self.pitch_range or pitch_range
if pitch_range is not None:
    pitch = self._fit_pitch_to_pitch_range(
        pitch, pitch_range,
    )
return pitch

### PRIVATE METHODS ###

def _constrain_interval(self, current_pitch, previous_pitch):
    if previous_pitch is None or not self.leap_constraint:
        return current_pitch
    maximum_leap = self.leap_constraint.semitones
    #semitones = float(current_pitch) - float(previous_pitch)
    semitones = float(previous_pitch) - float(current_pitch)
    if maximum_leap < semitones:  # descent
        current_pitch = current_pitch.transpose(12)
    elif semitones < -maximum_leap:  # ascent
        current_pitch = current_pitch.transpose(-12)
    return current_pitch

def _fit_pitch_to_pitch_range(self, pitch, pitch_range):
    while pitch <= pitch_range.start_pitch and \
        pitch not in pitch_range:
        pitch = pitch.transpose(12)
    while pitch_range.stop_pitch <= pitch and \
        pitch not in pitch_range:
        pitch = pitch.transpose(-12)
    assert pitch in pitch_range, \
        (pitch, pitch.octave_number, pitch_range)
    return pitch

def _get_pitch(self,
                pitch_class, registration, seed):
    octavations = self.octavations or self._default_octavations
    octave = octavations[seed]
    pitch = pitchtools.NamedPitch(pitch_class, octave)
    pitch_range = pitchtools.PitchRange('[C0, C8]')
    pitch = self._fit_pitch_to_pitch_range(pitch, pitch_range)
    pitch = registration([pitch])[0]
    pitch = pitchtools.NamedPitch(pitch)
    return pitch

def _get_pitch_class(self,
                      attack_point_signature, pitch_choices, 
                      previous_pitch_class, seed, 
                      557)
pitch_class = pitch_choices[seed]
pitch_class = pitchtools.NamedPitchClass(pitch_class)
if pitch_choices and \
  1 < len(set(pitch_choices)) and \
  self.forbid_repetitions:
  if self.pitch_application_rate == 'phrase':
    if attack_point_signature.is_first_of_phrase:
      while float(pitch_class) == float(previous_pitch_class):
        seed += 1
        pitch_class = pitch_choices[seed]
        pitch_class = pitchtools.NamedPitchClass(pitch_class)
  elif self.pitch_application_rate == 'division':
    if attack_point_signature.is_first_of_division:
      while float(pitch_class) == float(previous_pitch_class):
        seed += 1
        pitch_class = pitch_choices[seed]
        pitch_class = pitchtools.NamedPitchClass(pitch_class)
  else:
    while float(pitch_class) == float(previous_pitch_class):
      seed += 1
      pitch_class = pitch_choices[seed]
      pitch_class = pitchtools.NamedPitchClass(pitch_class)
      return pitch_class

def _get_registration_
  (self, 
    attack_point_signature, 
    logical_tie, 
    phrase_seed, 
  ):
    import consort
    register_specifier = self.register_specifier
    if register_specifier is None:
      register_specifier = consort.RegisterSpecifier()
      register = register_specifier.find_register(
        attack_point_signature, 
        seed=phrase_seed, 
      )
    register_spread = self.register_spread
    if register_spread is None:
      register_spread = 6
      registration = \n        pitchtools.Registration([ 
          ('[C0, C4]', register), 
          ('[C4, C8]', register + register_spread), 
        ]) 
      return registration

def _initialize_leap_constraint(self, leap_constraint):
  if leap_constraint is not None:
    leap_constraint = pitchtools.NumberedInterval(leap_constraint)
    leap_constraint = abs(leap_constraint)
    self._leap_constraint = leap_constraint
def _initialize_octavations(self, octavations):
    if octavations is not None:
        assert octavations
        assert all(isinstance(x, int) for x in octavations)
        octavations = datastructuretools.CyclicTuple(octavations)
        self._octavations = octavations

def _initialize_pitch_classes(self, pitch_classes):
    pitch_classes = pitchtools.PitchClassSegment(
        items=pitch_classes,
        item_class=pitchtools.NumberedPitchClass,
    )
    pitch_classes = datastructuretools.CyclicTuple(pitch_classes)
    self._pitch_classes = pitch_classes

def _initialize_pitch_range(self, pitch_range):
    if pitch_range is not None:
        assert isinstance(pitch_range, pitchtools.PitchRange)
        self._pitch_range = pitch_range

def _initialize_registerSpecifier(self, registerSpecifier):
    import consort
    if registerSpecifier is not None:
        assert isinstance(registerSpecifier, consort.RegisterSpecifier)
        self._registerSpecifier = registerSpecifier

def _initialize_registerSpread(self, registerSpread):
    if registerSpread is not None:
        registerSpread = int(registerSpread)
        assert 0 <= registerSpread < 12
        self._registerSpread = registerSpread

### PUBLIC METHODS ###

def transpose(self, expr):
    pitchSpecifier = self.pitchSpecifier
    if pitchSpecifier is not None:
        pitchSpecifier = pitchSpecifier.transpose(expr)
        registerSpecifier = self.registerSpecifier
        if registerSpecifier is not None:
            registerSpecifier = registerSpecifier.transpose(expr)
        return new(
            self,
            pitchSpecifier=pitchSpecifier,
            registerSpecifier=registerSpecifier,
        )

### PUBLIC PROPERTIES ###

@property
def leapConstraint(self):
    return self._leapConstraint
@property
def octavations(self):
    return self._octavations

@property
def pitch_range(self):
    return self._pitch_range

@property
def registerSpecifier(self):
    return self._register_specifier

@property
def registerSpread(self):
    return self._register_spread

A.34  consort.tools.PitchHandler

class PitchHandler(HashCachingObject):
    ### CLASS VARIABLES ###
    __slots__ = ('
        '_deviations',
        '_forbid_repetitions',
        '_grace_expressions',
        '_logical_tie_expressions',
        '_pitch_application_rate',
        '_pitch_operation_specifier',
        '_pitch_specifier',
        '_pitches_are_nonsemantic',
    )

    ### INITIALIZER ###
    def __init__(

self,
  deviations=\texttt{None},
  forbid_repetitions=\texttt{None},
  grace_expressions=\texttt{None},
  logical_tie_expressions=\texttt{None},
  pitch_application_rate=\texttt{None},
  pitch_specifier=\texttt{None},
  pitch_operation_specifier=\texttt{None},
  pitches_are_nonsemantic=\texttt{None},
):  
  \texttt{HashCachingObject.__init__(self)}
  self._initialize_deviations(deviations)
  self._initialize_forbid_repetitions(forbid_repetitions)
  self._initialize_grace_expressions(grace_expressions)
  self._initialize_logical_tie_expressions(logical_tie_expressions)
  self._initialize_pitch_application_rate(pitch_application_rate)
  self._initialize_pitch_specifier(pitch_specifier)
  self._initialize_pitch_operation_specifier(pitch_operation_specifier)
  if pitches_are_nonsemantic is \texttt{None}:  
    pitches_are_nonsemantic = \texttt{bool(pitches_are_nonsemantic)}
  self._pitches_are_nonsemantic = pitches_are_nonsemantic

### SPECIAL METHODS ###

\@abc.abstractmethod
def __call__(
  self,
  timewise_seed,
  logical_tie,
  attack_point_signature,
  phrase_seed,
  pitch_range,
  previous_pitch,
  seed,
  transposition,
):  
  raise \texttt{NotImplementedError}

### PRIVATE METHODS ###

def _apply_logical_tie_expression(
  self,
  logical_tie,
  pitch_range,
  seed,
):  
  if self.logical_tie_expressions:  
    logical_tie_expression = self.logical_tie_expressions[seed]
    if logical_tie_expression is \texttt{None}:
      logical_tie_expression(self.logical_tie,  
                             pitch_range=pitch_range,
  )
def _apply_deviation(self, pitch, seed):
    if self.deviations:
        deviation = self.deviations[seed]
        if isinstance(deviation, pitchtools.NumberedInterval):
            if deviation != 0:
                pitch = pitchtools.NumberedPitch(pitch)
                pitch = pitch.transpose(deviation)
                pitch = pitchtools.NamedPitch(pitch)
        elif isinstance(deviation, pitchtools.NamedInterval):
            pitch = pitch.transpose(deviation)
        return pitch
    else:
        return pitch

@staticmethod
def _get_timewise_seed(timewise_seeds_by_musicSpecifier, musicSpecifier):
    if musicSpecifier in timewise_seeds_by_musicSpecifier:
        timewise_seeds_by_musicSpecifier[musicSpecifier] += 1
    else:
        timewise_seeds_by_musicSpecifier[musicSpecifier] = 0
    return timewise_seeds_by_musicSpecifier[musicSpecifier]

@staticmethod
def _get_grace_logical_ties(logical_tie):
    logical_ties = []
    head = logical_tie.head
    previous_leaf = inspect_(head).get_leaf(-1)
    if previous_leaf is None:
        return logical_ties
    grace_containers = inspect_(previous_leaf).get_grace_containers('after')
    if grace_containers:
        grace_container = grace_containers[0]
        for logical_tie in iterate(grace_container).by_logical_tie(pitched=True):
            logical_ties.append(logical_tie)
    return logical_ties

@staticmethod
def _get_instrument(logical_tie, musicSpecifier):
    if musicSpecifier.instrument is not None:
        return musicSpecifier.instrument
    component = logical_tie.head
    prototype = instrumenttools.Instrument
    instrument = inspect_(component).get_effective(prototype)
    return instrument

@staticmethod
def _get_phrase_seed(attack_point_signature, music_specifier, phrase_seeds, voice):
    if attack_point_signature.is_first_of_phrase:
        if (voice, music_specifier) not in phrase_seeds:
            phrase_seed = (music_specifier.seed or 0) - 1
            phrase_seeds[(voice, music_specifier)] = phrase_seed
        phrase_seed = phrase_seeds[(voice, music_specifier)]
    return phrase_seed

def _get_pitch_choices(self, logical_tie, music_specifier, pitch_choice_timespans_by_music_specifier, segment_duration):
    if music_specifier not in pitch_choice_timespans_by_music_specifier:
        pitch_handler = music_specifier.pitch_handler
        pitch_specifier = pitch_handler.pitch_specifier
        operation_specifier = pitch_handler.pitch_operation_specifier
        pitch_choice_timespans = PitchHandler.get_pitch_choice_timespans(
            pitch_specifier=pitch_specifier,
            operation_specifier=operation_specifier,
            duration=segment_duration,
        )
        pitch_choice_timespans_by_music_specifier[music_specifier] = \
            pitch_choice_timespans
    timespans = pitch_choice_timespans_by_music_specifier[
        music_specifier]
    assert len(timespans)
    start_offset = logical_tie.get_timespan().start_offset
    # TODO: "overlapping" should include "starting at"
    found_timespans = \
        timespans.find_timespans_overlapping_offset(start_offset)
    found_timespans += \
        timespans.find_timespans_starting_at(start_offset)
    timespan = found_timespans[0]
    pitch_choices = timespan.annotation
    return pitch_choices

@staticmethod
def _get_pitch_range(instrument, logical_tie):
    prototype = pitchtools.PitchRange
    component = logical_tie.head
    pitch_range = inspect_(component).get_effective(prototype)
    if pitch_range is None and instrument is not None:
pitch_range = instrument.pitch_range
return pitch_range

@staticmethod
def _get_previous_pitch(
music_specifier,
previous_pitch_by_music_specifier,
voice,
):
key = (voice, music_specifier)
if key not in previous_pitch_by_music_specifier:
    previous_pitch_by_music_specifier[key] = None
previous_pitch = previous_pitch_by_music_specifier[key]
return previous_pitch

@staticmethod
def _get_pitch_seed(
attack_point_signature,
music_specifier,
pitch_application_rate,
pitch_seeds_by_music_specifier,
pitch_seeds_by_voice,
voice,
):
if music_specifier not in pitch_seeds_by_music_specifier:
    seed = (music_specifier.seed or 0) - 1
    pitch_seeds_by_music_specifier[music_specifier] = seed
    pitch_seeds_by_voice[voice] = seed
if pitch_application_rate == 'phrase':
    if attack_point_signature.is_first_of_phrase:
        pitch_seeds_by_music_specifier[music_specifier] += 1
        seed = pitch_seeds_by_music_specifier[music_specifier]
        pitch_seeds_by_voice[voice] = seed
    else:
        seed = pitch_seeds_by_voice[voice]
elif pitch_application_rate == 'division':
    if attack_point_signature.is_first_of_division:
        pitch_seeds_by_music_specifier[music_specifier] += 1
        seed = pitch_seeds_by_music_specifier[music_specifier]
        pitch_seeds_by_voice[voice] = seed
    else:
        seed = pitch_seeds_by_voice[voice]
else:
    pitch_seeds_by_music_specifier[music_specifier] += 1
    seed = pitch_seeds_by_music_specifier[music_specifier]
return seed

@staticmethod
def _get_sounding_pitch(
instrument,
pitch_handler,
):
if not instrument:
    return pitchtools.NamedPitch("c'")
sounding_pitch = instrument.sounding_pitch_of_written_middle_c
transposition_is_non_octave = sounding_pitch.named_pitch_class != \
    pitchtools.NamedPitchClass('c')
if transposition_is_non_octave:
    if pitch_handler.pitches_are_nonsemantic:
        return pitchtools.NamedPitch('c''')
    return sounding_pitch
return None

def _initialize_deviations(self, deviations):
    if deviations is not None:
        if not isinstance(deviations, collections.Sequence):
            deviations = (deviations,)
        assert len(deviations)
        intervals = []
        for interval in deviations:
            if isinstance(interval, (int, float)):
                interval = pitchtools.NumberedInterval(interval)
            elif isinstance(interval, str):
                interval = pitchtools.NamedInterval(interval)
            elif isinstance(interval, pitchtools.Interval):
                pass
            else:
                interval = pitchtools.NumberedInterval(interval)
            intervals.append(interval)
        deviations = datastructuretools.CyclicTuple(intervals)
        self._deviations = deviations

def _initialize_forbid_repetitions(self, forbid_repetitions):
    if forbid_repetitions is not None:
        forbid_repetitions = bool(forbid_repetitions)
        self._forbid_repetitions = forbid_repetitions

def _initialize_grace_expressions(self, grace_expressions):
    import consort
    if grace_expressions is not None:
        prototype = consort.LogicalTieExpression
        assert grace_expressions, grace_expressions
        assert all(isinstance(_, prototype) \ 
                   for _ in grace_expressions), \
        grace_expressions = datastructuretools.CyclicTuple( 
            grace_expressions,
        )
        self._grace_expressions = grace_expressions

def _initialize_logical_tie_expressions(self, logical_tie_expressions):
    import consort
    if logical_tie_expressions is not None:
        prototype = (consort.LogicalTieExpression, type(None))
        assert logical_tie_expressions, logical_tie_expressions
        assert all(isinstance(_, prototype) \ 
                   for _ in logical_tie_expressions), \
        logical_tie_expressions
logical_tie_expressions = datastructuretools.CyclicTuple(
    logical_tie_expressions,
)
self._logical_tie_expressions = logical_tie_expressions

def _initialize_pitch_application_rate(self, pitch_application_rate):
    assert pitch_application_rate in (
        None, 'logical_tie', 'division', 'phrase',
    )
    self._pitch_application_rate = pitch_application_rate

def _initialize_pitch_operationSpecifier(self, pitch_operationSpecifier):
    import consort
    if pitch_operationSpecifier is not None:
        prototype = consort.PitchOperationSpecifier
        if not isinstance(pitch_operationSpecifier, prototype):
            pitch_operationSpecifier = consort.PitchOperationSpecifier(
                pitch_operations=pitch_operationSpecifier,
            )
        self._pitch_operationSpecifier = pitch_operationSpecifier

def _initialize_pitchSpecifier(self, pitchSpecifier):
    import consort
    if pitchSpecifier is not None:
        if not isinstance(pitchSpecifier, consort.PitchSpecifier):
            pitchSpecifier = consort.PitchSpecifier(pitchSpecifier)
        self._pitchSpecifier = pitchSpecifier

def _process_logical_tie(self, logical_tie, pitch, pitch_range, seed):
    for leaf in logical_tie:
        leaf.written_pitch = pitch
    grace_logical_ties = self._get_grace_logical_ties(logical_tie)
    if str(pitch.accidental) and grace_logical_ties:
        leaf.note_head.is_forced = True
    self._apply_logical_tie_expression(
        logical_tie,
        seed=seed,
        pitch_range=pitch_range,
    )
    for i, grace_logical_tie in enumerate(grace_logical_ties, seed):
        for leaf in grace_logical_tie:
            leaf.written_pitch = pitch
        if self.grace_expressions:
            grace_expression = self.grace_expressions[i]
            grace_expression(grace_logical_tie)

@staticmethod
def _process_session(segment_maker):
    import consort
    maker = consort.SegmentMaker
    segment_duration = segment_maker.measure_offsets[-1]
    attack_point_map = segment_maker.attack_point_map
    pitch_choice_timespans_by_music_specifier = {}
seed_session = consort.SeedSession()
for logical_tie in attack_point_map:
    music_specifier = maker.logical_tie_to_musicSpecifier(logical_tie)
    if not music_specifier or not music_specifier.pitch_handler:
        continue
    pitch_handler = music_specifier.pitch_handler
    attack_point_signature = attack_point_map[logical_tie]
    application_rate = pitch_handler.pitch_application_rate
    voice = consort.SegmentMaker.logical_tie_to_voice(logical_tie)
    seed_session(
        application_rate,
        attack_point_signature,
        music_specifier,
        voice,
    )
    previous_pitch = pitch_handler._get_previous_pitch(
        music_specifier,
        previous_pitch_by_music_specifier,
        voice,
    )
    pitch_choices = pitch_handler._get_pitch_choices(
        logical_tie,
        music_specifier,
        pitch_choice_timespans_by_music_specifier,
        segment_duration,
    )
    pitch = pitch_handler(
        attack_point_signature,
        logical_tie,
        music_specifier,
        pitch_choices,
        previous_pitch,
        seed_session,
    )
    pitch_handler._set_previous_pitch(
        attack_point_signature,
        music_specifier,
        pitch,
        pitch_handler.pitch_application_rate,
        previous_pitch_by_music_specifier,
        voice,
    )
    pitch_handler._apply_transposition(
        attack_point_signature,
        logical_tie,
        music_specifier,
        pitch_handler,
        )
    instrument = pitch_handler._get_instrument(
        logical_tie,
        music_specifier,
    )
    pitch_range = pitch_handler._get_pitch_range(
        instrument,
logical_tie,
    )
pitch_handler._process_logical_tie(  
    logical_tie,
    pitch,
    pitch_range,
    seed_session.current_unphrased_voicewise_logical_tie_seed,
    )

@staticmethod
def _apply_transposition(  
    attack_point_signature,
    logical_tie,
    music_specifier,
    pitch_handler,
    ):
    import consort
    if not attack_point_signature.is_first_of_phrase:
        return
    voice = consort.SegmentMaker.logical_tie_to_voice(logical_tie)
    instrument = PitchHandler._get_instrument(    
        logical_tie, music_specifier)
    sounding_pitch = PitchHandler._get_sounding_pitch(    
        instrument, pitch_handler)
    if pitch_handler and pitch_handler.pitches_are_nonsemantic:
        sounding_pitch = pitchtools.NamedPitch('C4')
    if sounding_pitch is None:
        sounding_pitch = pitchtools.NamedPitch('C4')
    if sounding_pitch == pitchtools.NamedPitch('C4'):
        return
    phrase = consort.SegmentMaker.logical_tie_to_phrase(logical_tie)
    transposition_command = indicatortools.LilyPondCommand(    
        'transpose {!s} c'''.format(sounding_pitch),
        format_slot='before',
    )
    print(    
        ' transposing',
        voice.name,
        voice.index(phrase),
        sounding_pitch,
    )
    attach(transposition_command, phrase)

@staticmethod
def _set_previous_pitch(  
    attack_point_signature,
    music_specifier,
    pitch,
    pitch_application_rate,
    previous_pitch_by_music_specifier,
    voice,
    ):
    key = (voice, music_specifier)
    if pitch_application_rate == 'phrase':
if attack_point_signature.is_first_of_phrase:
    previous_pitch_by_music_specifier[key] = pitch
elif pitch_application_rate == 'division':
    if attack_point_signature.is_first_of_division:
        previous_pitch_by_music_specifier[key] = pitch
else:
    previous_pitch_by_music_specifier[key] = pitch

### PUBLIC METHODS ###

@staticmethod
def get_pitch_choice_timespans(
pitch_specifier=None,
operation_specifier=None,
duration=1,
):
    r'''Get pitch expression timespans.

    ::

    >>> import consort
    >>> pitch_specifier = consort.PitchSpecifier(
    ...     pitch_segments=(
    ...         "c' e' g'",
    ...         "fs' g' a'",
    ...         "b d",
    ...     ),
    ...     ratio=(1, 2, 3),
    ... )
    >>> operation_specifier = consort.PitchOperationSpecifier(
    ...     pitch_operations=(
    ...         pitchtools.PitchOperation(
    ...             pitchtools.Rotation(1),
    ...             pitchtools.Transposition(1),
    ...         ),
    ...         None,
    ...         pitchtools.PitchOperation(
    ...             pitchtools.Rotation(-1),
    ...             pitchtools.Transposition(-1),
    ...         ))
    ...     ),
    ...     ratio=(1, 2, 1),
    ... )
    >>> timespans = consort.PitchHandler.get_pitch_choice_timespans(
    ...     pitch_specifier=pitch_specifier,
    ...     operation_specifier=operation_specifier,
    ...     duration=12,
    ... )
    >>> print(format(timespans))
    consort.tools.TimespanCollection(
        [timespantools.AnnotatedTimespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(2, 1),
        ]
    ,)
annotation=datastructuretools.CyclicTuple(
    [
        pitchtools.NamedPitch("df'"),
        pitchtools.NamedPitch('gf'),
        pitchtools.NamedPitch('bf'),
    ]
),
),
timespantools.AnnotatedTimespan(
    start_offset=durationtools.Offset(2, 1),
    stop_offset=durationtools.Offset(3, 1),
    annotation=datastructuretools.CyclicTuple(
        [
            pitchtools.NamedPitch("g'"),
            pitchtools.NamedPitch("e'"),
            pitchtools.NamedPitch("f'"),
        ]
    ),
),
timespantools.AnnotatedTimespan(
    start_offset=durationtools.Offset(3, 1),
    stop_offset=durationtools.Offset(6, 1),
    annotation=datastructuretools.CyclicTuple(
        [
            pitchtools.NamedPitch("fs'"),
            pitchtools.NamedPitch("g'"),
            pitchtools.NamedPitch("a'"),
        ]
    ),
),
timespantools.AnnotatedTimespan(
    start_offset=durationtools.Offset(6, 1),
    stop_offset=durationtools.Offset(9, 1),
    annotation=datastructuretools.CyclicTuple(
        [
            pitchtools.NamedPitch('b'),
            pitchtools.NamedPitch('d'),
        ]
    ),
),
timespantools.AnnotatedTimespan(
    start_offset=durationtools.Offset(9, 1),
    stop_offset=durationtools.Offset(12, 1),
    annotation=datastructuretools.CyclicTuple(
        [
            pitchtools.NamedPitch('as'),
            pitchtools.NamedPitch("fss'"),
        ]
    ),
),
)

Returns timespans.
import consort
duration = durationtools.Duration(duration)
pitch_specifier = pitch_specifier or consort.PitchSpecifier()
operation_specifier = operation_specifier or 
    consort.PitchOperationSpecifier()
pitch_choice_timespans = consort.TimespanCollection()
pitch_timespans = pitch_specifier.get_timespans(duration)
operation_timespans = operation_specifier.get_timespans(duration)
offsets = set()
offsets.update(pitch_timespans.all_offsets)
offsets.update(operation_timespans.all_offsets)
offsets = tuple(sorted(offsets))
for start_offset, stop_offset in sequencetools.iterate_sequence_nwise(
    offsets):
    timespan = timespantools.Timespan(
        start_offset=start_offset,
        stop_offset=stop_offset,
    )
pitch_timespan = \
    pitch_timespans.find_timespans_intersecting_timespan(
        timespan)[0]
pitches = pitch_timespan.annotation
operation_timespan = \
    operation_timespans.find_timespans_intersecting_timespan(
        timespan)[0]
operation = operation_timespan.annotation
if operation is not None:
    pitches = operation(pitches)
pitch_choice_timespan = timespantools.CyclicTuple(pitches)
pitch_choice_timespan = timespantools.AnnotatedTimespan(
    annotation=pitches,
    start_offset=start_offset,
    stop_offset=stop_offset,
)
pitch_choice_timespans.insert(pitch_choice_timespan)
return pitch_choice_timespans

def transpose(self, expr):
    import consort
    pitch_specifier = self.pitch_specifier or consort.PitchSpecifier(
        pitch_segments='C4',
    )
pitch_specifier = pitch_specifier.transpose(expr)
    return new(self, pitch_specifier=pitch_specifier)

### PUBLIC PROPERTIES ###

@property
def deviations(self):
    return self._deviations

@property
def forbid_repetitions(self):
A.35 \texttt{consort.tools.PitchOperationSpecifier}

```python
class PitchOperationSpecifier(abctools.AbjadValueObject):
    r'''A operation specifier.

    .. container:: example

        >>> import consort
        >>> pitch_operation_specifier = consort.PitchOperationSpecifier(
        ...     pitch_operations=(
        ...         pitchtools.PitchOperation((
        ...             pitchtools.Rotation(1),
        ...             pitchtools.Transposition(1),
        ...         ),
        ...         None,
        ...         pitchtools.PitchOperation((
        ...         ))),
        ...         None),
```
... pitchtools.Rotation(-1),
... pitchtools.Transposition(-1),
... )
... ),
... ratio=(1, 2, 1),
... )

>>> print(format(pitch_operationSpecifier))
consort.tools.PitchOperationSpecifier(
pitch_operations=(
pitchtools.PitchOperation(
  operators=(
pitchtools.Rotation(
    index=1,
    transpose=True,
  ),
pitchtools.Transposition(
    index=1,
  ),
  ),
  None,
pitchtools.PitchOperation(
  operators=(
pitchtools.Rotation(
    index=-1,
    transpose=True,
  ),
pitchtools.Transposition(
    index=-1,
  ),
  ),
  ratio=mathtools.Ratio((1, 2, 1)),
  )
  ),
'.'/'.$

### CLASS VARIABLES ###

__slots__ =
  '_is_cumulative',
  '_pitch_operations',
  '_ratio',

### INITIALIZER ###

def __init__(
  self,
pitch_operations=None,
ratio=None,
is_cumulative=None,
):
if pitch_operations is not None:
    if not isinstance(pitch_operations, collections.Sequence):
        pitch_operations = (pitch_operations,)

    prototype = (pitchtools.PitchOperation, type(None),)
    coerced_pitch_operations = []
    for x in pitch_operations:
        if not isinstance(x, prototype):
            x = pitchtools.PitchOperation(x)
        coerced_pitch_operations.append(x)
    pitch_operations = tuple(coerced_pitch_operations)

    assert len(pitch_operations)

if pitch_operations and not ratio:
    ratio = [1] * len(pitch_operations)

if ratio is not None:
    ratio = mathtools.Ratio([abs(_.) for _ in ratio])
    assert len(ratio) == len(pitch_operations)

if is_cumulative is not None:
    is_cumulative = bool(is_cumulative)
    self._is_cumulative = is_cumulative
    self._pitch_operations = pitch_operations
    self._ratio = ratio

### PUBLIC METHODS ###

def get_timespans(self, stop_offset):
    r'''Gets pitch expr timespans.

    .. container:: example

    ::

        >>> timespans = pitch_operation_specifier.get_timespans(10)
        >>> print(format(timespans))
        consort.tools.TimespanCollection(
            [ timespantools.AnnotatedTimespan(
                 start_offset=durationtools.Offset(0, 1),
                 stop_offset=durationtools.Offset(5, 2),
                 annotation=pitchtools.PitchOperation(
                     operators=(
                         pitchtools.Rotation(
                             index=1,
                             transpose=True,
                         ),
                         pitchtools.Transposition(
                             index=1,
                         ),
                     ),
                 ),
            ]
        )
timespan = AnnotatedTimespan(
    start_offset=offset(5, 2),
    stop_offset=offset(15, 2),
),
    AnnotatedTimespan(
        start_offset=offset(15, 2),
        stop_offset=offset(10, 1),
        annotation=PitchOperation(
            operators=(
                Rotation(index=-1, transpose=True),
                Transposition(),
            ),
        ),
    ),
)

.. container:: example

    >>> pitch_operation_specifier = PitchOperationSpecifier(
    ...     is_cumulative=True,
    ...     pitch_operations=(
    ...         Rotation(1),
    ...         Transposition(1),
    ...         Inversion(),
    ...         None,
    ...     ),
    ...     ratio=(1, 2, 3, 1),
    ... )
    >>> timespans = pitch_operation_specifier.get_timespans(10)
    >>> print(format(timespans))
    consort.tools.TimespanCollection(
        [AnnotatedTimespan(
            start_offset=offset(0, 1),
            stop_offset=offset(10, 7),
            annotation=PitchOperation(
                operators=(
                    Rotation(index=1, transpose=True),
                ),
            ),
        ),
    )
timespantools.AnnotatedTimespan(
    start_offset=durationtools.Offset(10, 7),
    stop_offset=durationtools.Offset(30, 7),
    annotation=pitchtools.PitchOperation(
        operators=(
            pitchtools.Rotation(
                index=1,
                transpose=True,
            ),
            pitchtools.Transposition(
                index=1,
            ),
            pitchtools.Inversion(),
        ),
    ),
),

),
timespantools.AnnotatedTimespan(
    start_offset=durationtools.Offset(30, 7),
    stop_offset=durationtools.Offset(60, 7),
    annotation=pitchtools.PitchOperation(
        operators=(
            pitchtools.Rotation(
                index=1,
                transpose=True,
            ),
            pitchtools.Transposition(
                index=1,
            ),
            pitchtools.Inversion(),
        ),
    ),
),

timespantools.AnnotatedTimespan(
    start_offset=durationtools.Offset(60, 7),
    stop_offset=durationtools.Offset(10, 1),
    annotation=pitchtools.PitchOperation(
        operators=(
            pitchtools.Rotation(
                index=1,
                transpose=True,
            ),
            pitchtools.Transposition(
                index=1,
            ),
            pitchtools.Inversion(),
        ),
    ),
)
]

Returns timespan collection.
',''

import consort
timespans = consort.TimeSpanCollection()
if not self.ratio or not self.pitch_operations:
    annotated_timespan = timespan_tools.AnnotatedTimespan(
        start_offset=0,
        stop_offset=stop_offset,
    ),
timespans.insert(annotated_timespan)
else:
    target_timespan = timespan_tools.TimeSpan(
        start_offset=0,
        stop_offset=stop_offset,
    )
divided_timespans = target_timespan.divide_by_ratio(self.ratio)
pitch_operation = pitchtools.PitchOperation()
for i, timespan in enumerate(divided_timespans):
    current_pitch_operation = self._pitch_operations[i]
    if self.is_cumulative:
        if current_pitch_operation:
            pitch_operation = pitchtools.PitchOperation(
                (pitch_operation.operators or ()) +
                current_pitch_operation.operators
            )
        else:
            pitch_operation = current_pitch_operation
    annotated_timespan = timespan_tools.AnnotatedTimespan(
        annotation=pitch_operation,
        start_offset=timespan.start_offset,
        stop_offset=timespan.stop_offset,
    )
timespans.insert(annotated_timespan)
return timespans

def rotate(self, rotation):
    r'''Rotates pitch operation specifier.
    ::

    >>> pitch_operation_specifier = consort.PitchOperationSpecifier(
    ...     pitch_operations=(
    ...         pitchtools.PitchOperation(
    ...             pitchtools.Rotation(1),
    ...             pitchtools.Transposition(1),
    ...         ),
    ...     ),
    ...     None,
    ...     pitchtools.PitchOperation(
    ...         pitchtools.Rotation(-1),
    ...         pitchtools.Transposition(-1),
    ...     ))
    ...     ratio=(1, 2, 1),
    ... )

    >>> rotated_specifier = pitch_operation_specifier.rotate(1)
    >>> print(format(rotated_specifier))
    consort.tools.PitchOperationSpecifier(
pitch_operations = (  
    pitchtools.PitchOperation(  
        operators=(  
            pitchtools.Rotation(  
                index=-1,  
                transpose=True,  
            ),  
            pitchtools.Transposition(  
                index=-1,  
            ),  
            pitchtools.PitchOperation(  
                operators=(  
                    pitchtools.Rotation(  
                        index=1,  
                        transpose=True,  
                    ),  
                    pitchtools.Transposition(  
                        index=1,  
                    ),  
                    None,  
                    ),  
                ratio=mathtools.Ratio((1, 1, 2)),  
            )  
        ),  
    ),  
)

Returns new pitch specifier.

rotation = int(rotation)
pitch_operations = sequencetools.rotate_sequence(  
    self.pitch_operations, rotation)

ratio = sequencetools.rotate_sequence(self.ratio, rotation)
return new(  
    self,  
    pitch_operations=pitch_operations,  
    ratio=ratio,  
)

# PUBLIC PROPERTIES #

@property
def is_cumulative(self):
    return self._is_cumulative

@property
def ratio(self):
    return self._ratio

@property
def pitch_operations(self):
    return self._pitch_operations
A.36  consort.tools.PitchSpecifier

```python
# -*- encoding: utf-8 -*-
from abjad import new
from abjad.tools import abctools
from abjad.tools import mathtools
from abjad.tools import pitchtools
from abjad.tools import sequencetools
from abjad.tools import timespantools

class PitchSpecifier(abctools.AbjadValueObject):
    r'''A pitch specifier.
    ::
    >>> import consort
    >>> pitch_specifier = consort.PitchSpecifier(
    ...     pitch_segments=('c' 'e' 'g',
    ...     'fs' 'gs',
    ...     'b',
    ... ),
    ...     ratio=(1, 2, 3),
    ... )
    >>> print(format(pitch_specifier))
    consort.tools.PitchSpecifier(
        pitch_segments=(
            pitchtools.PitchSegment(
                pitchtools.NamedPitch("c"),
                pitchtools.NamedPitch("e"),
                pitchtools.NamedPitch("g"),
            ),
            item_class=pitchtools.NamedPitch,
        ),
        pitchtools.PitchSegment(
            pitchtools.NamedPitch("fs"),
            pitchtools.NamedPitch("gs"),
        ),
        item_class=pitchtools.NamedPitch,
        pitchtools.PitchSegment(
            pitchtools.NamedPitch('b'),
        ),
        ratio=mathtools.Ratio((1, 2, 3)),
    )
    Pitch specifiers can be instantiated from a string of pitch names:
```

579
>>> pitch_specifier = consort.PitchSpecifier("c' e' g' a'")
>>> print(format(pitch_specifier))
consort.tools.PitchSpecifier(
    pitch_segments=(
        pitchtools.PitchSegment(
            (pitchtools.NamedPitch("c'"),
             pitchtools.NamedPitch("e'"),
             pitchtools.NamedPitch("g'"),
             pitchtools.NamedPitch("a'"),
             ),
             item_class=pitchtools.NamedPitch,
             ),
         ratio=mathtools.Ratio((1,)),
    )
)

Pitch specifiers can be instantiated from a single pitch:

>>> pitch_specifier = consort.PitchSpecifier(NamedPitch("ds'"))
>>> print(format(pitch_specifier))
consort.tools.PitchSpecifier(
    pitch_segments=(
        pitchtools.PitchSegment(
            (pitchtools.NamedPitch("ds'"),
             ),
             item_class=pitchtools.NamedPitch,
             ),
         ratio=mathtools.Ratio((1,)),
    )
)

### CLASS VARIABLES ###

__slots__ = ('_pitch_segments',
             '_ratio',
)

### INITIALIZER ###

def __init__(
    self,
    pitch_segments=None,
    ratio=None,
):
if pitch_segments is not None:
    if isinstance(pitch_segments, pitchtools.Pitch):
        pitch_segments = pitchtools.PitchSegment([pitch_segments])
    elif isinstance(pitch_segments, str):
        pitch_segments = pitchtools.PitchSegment(pitch_segments)
    if isinstance(pitch_segments, pitchtools.PitchSegment):
        coerced_pitch_segments = []
        for pitch_segment in pitch_segments:
            pitch_segment = pitchtools.PitchSegment(
                pitch_segment,
                item_class=pitchtools.NamedPitch,
            )
            if not pitch_segment:
                pitch_segment = pitchtools.PitchSegment("c'")
            coerced_pitch_segments.append(pitch_segment)
        pitch_segments = tuple(coerced_pitch_segments)
        assert len(pitch_segments)
    if pitch_segments and not ratio:
        ratio = [1] * len(pitch_segments)
    if ratio is not None:
        ratio = mathtools.Ratio([abs(x) for x in ratio])
        assert len(ratio) == len(pitch_segments)
    self._pitch_segments = pitch_segments
    self._ratio = ratio

### PUBLIC METHODS ###

def get_timespans(self, stop_offset):
    r'''Gets pitch segment timespans.

    ::

    >>> pitch_specifier = consort.PitchSpecifier(
    ...     pitch_segments=('c' 'e' 'g''
    ...     "fs' g'",
    ...     "b",
    ... ),
    ...     ratio=(1, 2, 3),
    ... )
    >>> timespans = pitch_specifier.get_timespans(stop_offset=10)
    >>> print(format(timespans))
    consort.tools.TimespanCollection(
        [timespan tools. AnnotatedTimespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(5, 3),
            annotation=pitchtools.PitchSegment(
                pitchtools.NamedPitch("c'"),
            ),
            )]
```
161       pitchtools.NamedPitch("e'",
162       pitchtools.NamedPitch("g'"),
163                       item_class=pitchtools.NamedPitch,
164       ),
165     timespantools.AnnotatedTimespan(
166         start_offset=durationtools.Offset(5, 3),
167         stop_offset=durationtools.Offset(5, 1),
168         annotation=pitchtools.PitchSegment(
169            pitchtools.NamedPitch("fs'",
170            pitchtools.NamedPitch("g'",
171                 item_class=pitchtools.NamedPitch,
172               ),
173         ),
174     ),
175   timespantools.AnnotatedTimespan(
176         start_offset=durationtools.Offset(5, 1),
177         stop_offset=durationtools.Offset(10, 1),
178         annotation=pitchtools.PitchSegment(
179            pitchtools.NamedPitch('b'),
180                 item_class=pitchtools.NamedPitch,
181             ),
182       )
183   )
184
185   import consort
186   timespans = consort.TimespanCollection()
187   if not self.ratio or not self.pitch_segments:
188       pitch_segment = pitchtools.PitchSegment("c'")
189       annotated_timespan = timespantools.AnnotatedTimespan(
190            annotation=pitch_segment,
191            start_offset=0,
192            stop_offset=stop_offset,
193         ),
194       timespans.insert(annotated_timespan)
195   else:
196       target_timespan = timespantools.Timespan(
197            start_offset=0,
198            stop_offset=stop_offset,
199         )
200       divided_timespans = target_timespan.divide_by_ratio(self.ratio)
201       for timespan in enumerate(divided_timespans):
202         pitch_segment = self._pitch_segments[i]
203         annotated_timespan = timespantools.AnnotatedTimespan(
204            annotation=pitch_segment,
205            start_offset=timespan.start_offset,
206            stop_offset=timespan.stop_offset,
207         )
```
```python
timespans.insert(annotated_timespan)

def rotate(self, rotation):
    r'''Rotates pitch specifier.
    ::
    >>> pitch_specifier = consort.PitchSpecifier(
    ...     pitch_segments=(
    ...         "c' e' g'",
    ...         "fs' gs'",
    ...         "b",
    ...     ),
    ...     ratio=(1, 2, 3),
    ... )
    >>> rotated_pitch_specifier = pitch_specifier.rotate(1)
    >>> print(format(rotated_pitch_specifier))
    consort.tools.PitchSpecifier(
        pitch_segments=(
            pitchtools.PitchSegment(
                (
                    pitchtools.NamedPitch('b'),
                ),
                item_class=pitchtools.NamedPitch,
            ),
            pitchtools.PitchSegment(
                (
                    pitchtools.NamedPitch("c"),
                    pitchtools.NamedPitch("f"),
                    pitchtools.NamedPitch("a"),
                ),
                item_class=pitchtools.NamedPitch,
            ),
            pitchtools.PitchSegment(
                (
                    pitchtools.NamedPitch("fs"),
                    pitchtools.NamedPitch("e"),
                ),
                item_class=pitchtools.NamedPitch,
            ),
        ),
        ratio=mathtools.Ratio((3, 1, 2)),
    )
    Returns new pitch specifier.
    '''
    rotation = int(rotation)
pitch_segments = tuple(
        _.rotate(rotation, transpose=True)
        for _ in self.pitch_segments)
pitch_segments = sequencetools.rotate_sequence(
            pitch_segments, rotation)
```
583
ratio = sequencetools.rotate_sequence(self.ratio, rotation)
return new(
    self,
    pitch_segments=pitch_segments,
    ratio=ratio,
)

def transpose(self, expr=0):
    r'''Transposes pitch specifier.
    ::
    >>> pitch_specifier = consort.PitchSpecifier(
    ...     pitch_segments=(
    ...         "c' e' g'",
    ...         "fs' gs'",
    ...         "b",
    ...     ),
    ...     ratio=(1, 2, 3),
    ... )
    >>> transposed_pitch_specifier = pitch_specifier.transpose('M2')
    >>> print(format(transposed_pitch_specifier))
    consort.tools.PitchSpecifier(
        pitch_segments=(
            pitchtools.PitchSegment(
                (pitchtools.NamedPitch("d'"),
                 pitchtools.NamedPitch("fs'"),
                 pitchtools.NamedPitch("a'"),
                 item_class=pitchtools.NamedPitch,
             ),
            pitchtools.PitchSegment(
                (pitchtools.NamedPitch("gs'"),
                 pitchtools.NamedPitch("as'"),
                 item_class=pitchtools.NamedPitch,
             ),
            pitchtools.PitchSegment(
                (pitchtools.NamedPitch("cs'"),
                 item_class=pitchtools.NamedPitch,
             ),
            ),
        )
    ratio=mathtools.Ratio((1, 2, 3)),
)
    Returns new pitch specifier.
    '''
    pitch_segments = (_ .transpose(expr) for _ in self.pitch_segments)
    return new(self, pitch_segments=pitch_segments)
### PUBLIC PROPERTIES ###

```python
@property
def ratio(self):
    return self._ratio
```

```python
@property
def pitch_segments(self):
    return self._pitch_segments
```

### CLASS VARIABLES ###

```python
__slots__ = ()
```

### INITIALIZER ###

```python
def __init__(self, items=None):
    datastructuretools.TypedList.__init__(self, items=items,
)
```

### PRIVATE PROPERTIES ###

```python
@property
def _attribute_manifest(self):
    from abjad.tools import systemtools
    return systemtools.AttributeManifest()
```

### PUBLIC METHODS ###

```python
def get_segment_desired_duration_in_seconds(self, segment_index, total_seconds):
    segment_proportions = self[segment_index]
    segment_total = sum(sequencetools.flatten_sequence(
        segment_proportions))
    ratio = durationtools.Multiplier(segment_total, self.total)
    desired_duration_in_seconds = ratio * total_seconds
    return desired_duration_in_seconds
```

### PUBLIC PROPERTIES ###

```python
@property
def total(self):
    return sum(sequencetools.flatten_sequence(self))
```


.. container:: example

    >>> import consort
    >>> expression = consort.RatioPartsExpression(
    ...     ratio=(1, 2, 1),
    ...     parts=(0, 2),
    ...     )
    >>> print(format(expression))
    consort.tools.RatioPartsExpression(
        parts=(0, 2),
        ratio=mathtools.Ratio((1, 2, 1)),
    )

.. container:: example

    >>> timespan = timespantools.Timespan(0, 4)
    >>> for x in expression(timespan):
    ...     x
    ...     Timespan(start_offset=Offset(1, 2), stop_offset=Offset(3, 4))
    Timespan(start_offset=Offset(5, 4), stop_offset=Offset(3, 2))

.. container:: example

    >>> expression = consort.RatioPartsExpression(
    ...     ratio=(1, 2, 1),
    ...     parts=(0, 2),
    ...     mask_timespan=timespantools.Timespan(
    ...         start_offset=(1, 4),
    ...     )
    ...     )
    >>> timespan = timespantools.Timespan(0, 4)
>>> for x in expression(timespan):
...    x
...  Timespan(start_offset=Offset(1, 4), stop_offset=Offset(1, 1))
  Timespan(start_offset=Offset(3, 1), stop_offset=Offset(4, 1))

### CLASS VARIABLES ###

__slots__ = 
  '_parts',
  '_ratio',
  '_mask_timespan',

### INITIALIZER ###

def __init__(self, parts=0, ratio=(1, 1), mask_timespan=None):
  if not isinstance(ratio, mathtools.Ratio):
    ratio = mathtools.Ratio(ratio)
  self._ratio = ratio
  if isinstance(parts, int):
    parts = (parts,)
  assert all(0 <= _ < len(ratio) for _ in parts)
  parts = tuple(sorted(set(parts)))
  self._parts = parts
  if mask_timespan is not None:
    assert isinstance(mask_timespan, timespantools.Timespan)
  self._mask_timespan = mask_timespan

### SPECIAL METHODS ###

def __call__(self, timespan):
  assert isinstance(timespan, timespantools.Timespan)
  divided_timespan = timespan.divide_by_ratio(self.ratio)
  timespans = timespantools.TimespanInventory()
  for part in self.parts:
    timespans.append(divided_timespan[part])
  if self.mask_timespan is not None:
    timespans & self.mask_timespan
  return timespans

### PUBLIC METHODS ###

@staticmethod
def from_sequence(sequence):
  r'''Creates a ratio parts expression from `sequence`.
>>> ratio = [-1, 2, -1, 1, -1]
>>> expression = consort.RatioPartsExpression.from_sequence(ratio)
>>> print(format(expression))
consort.tools.RatioPartsExpression(
    parts=(1, 3),
    ratio=mathtools.Ratio((1, 2, 1, 1, 1)),
)

Returns new ratio parts expression.

```
assert all(sequence)
assert len(sequence)
ratio = []
parts = []
for i, x in enumerate(sequence):
    if 0 < x:
        parts.append(i)
        ratio.append(abs(x))
result = RatioPartsExpression(
    parts=parts,
    ratio=ratio,
)
return result
```
>>> import consort
>>> register_inflection = consort.RegisterInflection(
...     inflections=(-6, 0, 9),
...     ratio=(2, 1),
... )
>>> print(format(register_inflection))
consort.tools.RegisterInflection(
    inflections=pitchtools.IntervalSegment(
        (pitchtools.NumberedInterval(-6),
         pitchtools.NumberedInterval(0),
         pitchtools.NumberedInterval(9),
        ),
        item_class=pitchtools.NumberedInterval,
    ),
    ratio=mathtools.Ratio((2, 1)),
)

>>> register_inflection(0)
NumberedInterval(-6)

>>> register_inflection((1, 3))
NumberedInterval(-3)

>>> register_inflection((1, 2))
NumberedInterval(-1)

>>> register_inflection((2, 3))
NumberedInterval(0)

>>> register_inflection(1)
NumberedInterval(9)

### CLASS VARIABLES ###

__slots__ = ('_inflections',
             '_ratio',
)
### INITIALIZER ###

```python
def __init__(self, inflections=(0, 0), ratio=(1,)):
    if isinstance(inflections, type(self)):
        expr = inflections
        self._inflections = expr.inflections
        self._ratio = expr.ratio
        return
    ratio = mathtools.Ratio([abs(x) for x in ratio])
    self._ratio = ratio
    inflections = pitchtools.IntervalSegment(
        inflections,
        item_class=pitchtools.NumberedInterval,
    )
    self._inflections = inflections
    assert len(inflections) == len(ratio) + 1
```

### SPECIAL METHODS ###

```python
def __call__(self, position):
    position = durationtools.Offset(position)
    if position < 0:
        position = durationtools.Offset(0)
    if 1 < position:
        position = durationtools.Offset(1)
    if position == 0:
        return self.inflections[0]
    elif position == 1:
        return self.inflections[-1]
    ratio_sum = sum(self.ratio)
    positions = [durationtools.Offset(x) / ratio_sum
                 for x in mathtools.cumulative_sums(self.ratio)]
    index = bisect.bisect(positions, position)
    position = float(position)
    x0 = float(positions[index - 1])
    x1 = float(positions[index])
    y0 = float(self.inflections[index - 1])
    y1 = float(self.inflections[index])
    dx = x1 - x0
    dy = y1 - y0
    m = float(dy) / float(dx)
    b = y0 - (m * x0)
    result = (position * m) + b
    result = pitchtools.NumberedInterval(int(result))
    return result
```

### PUBLIC METHODS ###

```python
def align(self):
```
Aligns all inflections to a minimum interval of zero.

```python
>>> import consort
>>> inflection = consort.RegisterInflection.zigzag().align()
>>> print(format(inflection))
consort.tools.RegisterInflection(
    inflections=pitchtools.IntervalSegment(
        pitchtools.NumberedInterval(0),
        pitchtools.NumberedInterval(9),
        pitchtools.NumberedInterval(3),
        pitchtools.NumberedInterval(12),
    ),
    item_class=pitchtools.NumberedInterval,
    ),
    ratio=mathtools.Ratio((1, 1, 1)),
)
```

Emits new register inflection.

```
minimum = sorted(self.inflections, key=lambda x: x.semitones)[0]
inflections = (_ - minimum for _ in self.inflections)
return new(self,
    inflections=inflections,
)
```

@staticmethod
def ascending(width=12):
    r'''Creates an ascending register inflection.
    ...
    minimum = sorted(self.inflections, key=lambda x: x.semitones)[0]
    inflections = (_ - minimum for _ in self.inflections)
    return new(self,
        inflections=inflections,
    )
```

Emits new register inflection.
```
import consort
half_width = abs(int(width / 2))
return consort.RegisterInflection(
    inflections=(0 - half_width, half_width),
)
@staticmethod
def descending(width=12):
    r'''Creates a descending register inflection.

    >>> import consort
    >>> inflection = consort.RegisterInflection.descending()
    >>> print(format(inflection))
    consort.tools.RegisterInflection(
        inflections=pitchtools.IntervalSegment(
            pitchtools.NumberedInterval(6),
            pitchtools.NumberedInterval(-6),
        ),
        ratio=mathtools.Ratio((1,)),
    )

    Emits new register inflection.

    '''
    return consort.RegisterInflection.ascending(width=width).invert()

def invert(self):
    r'''Inverts register inflection

    >>> import consort
    >>> inflection = consort.RegisterInflection.triangle().invert()
    >>> print(format(inflection))
    consort.tools.RegisterInflection(
        inflections=pitchtools.IntervalSegment(
            pitchtools.NumberedInterval(6),
            pitchtools.NumberedInterval(-6),
            pitchtools.NumberedInterval(6),
        ),
        ratio=mathtools.Ratio((1, 1)),
    )

    Emits new register inflection.

    '''
    return new(self,
        inflections=(-x for x in self.inflections),
    )
def reverse(self):
    r'''Reverses register inflection.
    ::
        >>> import consort
        >>> inflection = consort.RegisterInflection.zigzag().reverse()
        >>> print(format(inflection))
        consort.tools.RegisterInflection(
            inflections=pitchtools.IntervalSegment(
                pitchtools.NumberedInterval(6),
                pitchtools.NumberedInterval(-3),
                pitchtools.NumberedInterval(3),
                pitchtools.NumberedInterval(-6),
            ),
            item_class=pitchtools.NumberedInterval,
            ratio=mathtools.Ratio((1, 1, 1)),
        )
    Emits new register inflection.
    '''
    return new(self,
        inflections=reversed(self.inflections),
        ratio=reversed(self.ratio),
    )

def rotate(self, n=1):
    r'''Rotates register inflection by 'n'.
    ::
        >>> import consort
        >>> inflection = consort.RegisterInflection.zigzag().rotate(1)
        >>> print(format(inflection))
        consort.tools.RegisterInflection(
            inflections=pitchtools.IntervalSegment(
                pitchtools.NumberedInterval(6),
                pitchtools.NumberedInterval(-6),
                pitchtools.NumberedInterval(3),
                pitchtools.NumberedInterval(-3),
            ),
            item_class=pitchtools.NumberedInterval,
            ratio=mathtools.Ratio((1, 1, 1)),
        )
    Emits new register inflection.
    '''
    return new(self,
        inflections=sequencetools.rotate_sequence(self.inflections, n),
        ratio=sequencetools.rotate_sequence(self.ratio, n),
    )
@staticmethod
def triangle(width=12):
    r'''Creates a triangular register inflection.
    ::

>>> import consort
>>> inflection = consort.RegisterInflection.triangle()
>>> print(format(inflection))
consort.tools.RegisterInflection(
    inflections=pitchtools.IntervalSegment(
        (pitchtools.NumberedInterval(-6),
         pitchtools.NumberedInterval(6),
         pitchtools.NumberedInterval(-6),
        ),
        item_class=pitchtools.NumberedInterval,
    ),
    ratio=mathtools.Ratio((1, 1)),
)
::

>>> import consort
>>> inflection = consort.RegisterInflection.triangle(width=6)
>>> print(format(inflection))
consort.tools.RegisterInflection(
    inflections=pitchtools.IntervalSegment(
        (pitchtools.NumberedInterval(-3),
         pitchtools.NumberedInterval(3),
         pitchtools.NumberedInterval(-3),
        ),
        item_class=pitchtools.NumberedInterval,
    ),
    ratio=mathtools.Ratio((1, 1)),
)

Emits new register inflection.
'''
import consort
half_width = int(width / 2)
return consort.RegisterInflection(
    inflections=([-half_width, half_width, -half_width],
                 ratio=(1, 1)),
)

@staticmethod
def zigzag(width=12):
    r'''Creates a zigzag register inflection.
    ::
```python
>>> import consort
>>> inflection = consort.RegisterInflection.zigzag()
>>> print(format(inflection))
consort.tools.RegisterInflection(
    inflections=pitchtools.IntervalSegment(
        pitchtools.NumberedInterval(-6),
        pitchtools.NumberedInterval(3),
        pitchtools.NumberedInterval(-3),
        pitchtools.NumberedInterval(6),
    ),
    item_class=pitchtools.NumberedInterval,
    ratio=mathtools.Ratio((1, 1, 1)),
)
::

>>> import consort
>>> inflection = consort.RegisterInflection.zigzag(width=8)
>>> print(format(inflection))
consort.tools.RegisterInflection(
    inflections=pitchtools.IntervalSegment(
        pitchtools.NumberedInterval(-4),
        pitchtools.NumberedInterval(2),
        pitchtools.NumberedInterval(-2),
        pitchtools.NumberedInterval(4),
    ),
    item_class=pitchtools.NumberedInterval,
    ratio=mathtools.Ratio((1, 1, 1)),
)
::

Emits new register inflection.
```

```python
import consort
half_width = int(width / 2)
quarter_width = int(width / 4)
return consort.RegisterInflection(
    inflections=(
        -half_width,
        quarter_width,
        -quarter_width,
        half_width,
    ),
    ratio=(1, 1, 1),
)
```
return self._inflections

@property
def ratio(self):
    return self._ratio

A.40  consort.tools.RegisterInflectionInventory

class RegisterInflectionInventory(datastructuretools.TypedList):
    
    ### CLASS VARIABLES ###
    
    __slots__ = ()

    ### PRIVATE PROPERTIES ###

    @property
def _attribute_manifest(self):
        from abjad.tools import systemtools
        return systemtools.AttributeManifest()

    @property
def _item_callable(self):
        import consort
        return pitchtools.RegisterInflection

A.41  consort.tools.RegisterSpecifier

class RegisterSpecifier(abctools.AbjadValueObject):
    r'''
    A register specifier.
    
    ::

>>> import consort
>>> registerSpecifier = consort.RegisterSpecifier(
...     base_pitch=12,
...     division_inflections=(
...         consort.RegisterInflection(
...             inflections=(-6, 3, 6),
...             ratio=(1, 1),
...         ),
...     ),
... ),
>>> print(format(registerSpecifier))
consort.tools.RegisterSpecifier(
    base_pitch=pitchtools.NumberedPitch(12),
    division_inflections=consort.tools.RegisterInflectionInventory(
        ),
    phrase_inflections=consort.tools.RegisterInflectionInventory(
        ),
    segment_inflections=consort.tools.RegisterInflectionInventory(
        ]
)
>>> attack_point_signature = consort.AttackPointSignature(
...     division_position=0,
...     phrase_position=(1, 2),
...     segment_position=(4, 5),
... )
>>> registerSpecifier.find_register(attack_point_signature)
NumberedPitch(6)

### CLASS VARIABLES ###

__slots__ = ('_base_pitch',
             '_division_inflections',
             '_phrase_inflections',
             '_segment_inflections',
             )

### INITIALIZER ###

def __init__(
    self,
    base_pitch=None,
    division_inflections=None,
    phrase_inflections=None,
    segment_inflections=None,
):
    from consort.tools import RegisterInflectionInventory
    if isinstance(base_pitch, type(self)):
        expr = base_pitch
        self._base_pitch = expr.base_pitch
        self._division_inflections = expr.division_inflections
        self._phrase_inflections = expr.phrase_inflections
        self._segment_inflections = expr.segment_inflections
        return
    if base_pitch is not None:
        base_pitch = pitchtools.NumberedPitch(base_pitch)
        self._base_pitch = base_pitch
    if division_inflections is not None:
        if not isinstance(division_inflections, collections.Sequence):
            division_inflections = [division_inflections]
        division_inflections = RegisterInflectionInventory(
division_inflections)

self._division_inflections = division_inflections

if phrase_inflections is not None:
    if not isinstance(phrase_inflections, collections.Sequence):
        phrase_inflections = RegisterInflectionInventory(phrase_inflections)
    self._phrase_inflections = phrase_inflections

if segment_inflections is not None:
    if not isinstance(segment_inflections, collections.Sequence):
        segment_inflections = RegisterInflectionInventory(segment_inflections)
    self._segment_inflections = segment_inflections

### PUBLIC METHODS ###

def find_register(self, attack_point_signature, seed=0):
    division_position = attack_point_signature.division_position
    phrase_position = attack_point_signature.phrase_position
    segment_position = attack_point_signature.segment_position
    seed = int(seed)
    register = self.base_pitch
    if register is None:
        register = pitchtools.NumberedPitch(0)
    if self.division_inflections:
        index = seed % len(self.division_inflections)
        inflection = self.division_inflections[index]
        deviation = inflection(division_position)
        register = register.transpose(deviation)
    if self.phrase_inflections:
        index = seed % len(self.phrase_inflections)
        inflection = self.phrase_inflections[index]
        deviation = inflection(phrase_position)
        register = register.transpose(deviation)
    if self.segment_inflections:
        index = seed % len(self.segment_inflections)
        inflection = self.segment_inflections[index]
        deviation = inflection(segment_position)
        register = register.transpose(deviation)
    return register

def transpose(self, transposition):
    r'''Transposes register specifier.
    ::
    >>> register_specifier = consort.RegisterSpecifier(
    ...     base_pitch=12,
    ...     division_inflections={

599
... consort.RegisterInflection(
... inflections=(-6, 3, 6),
... ratio=(1, 1),
... ),
... phrase_inflections=
... consort.RegisterInflection(
... inflections=(3, -3),
... ratio=(1,),
... ),
... segment_inflections=
... consort.RegisterInflection(
... inflections=(-12, -9, 0, 12),
... ratio=(3, 2, 1),
... ),
... )
>>> transposedSpecifier = registerSpecifier.transpose(-6)
>>> print(format(transposedSpecifier))
consort.tools.RegisterSpecifier(
  base_pitch=pitchtools.NumberedPitch(6),
  division_inflections=consort.tools.RegisterInflectionInventory(
    [      consort.tools.RegisterInflection(
      inflections=pitchtools.IntervalSegment(
        (      pitchtools.NumberedInterval(-6),
          pitchtools.NumberedInterval(3),
          pitchtools.NumberedInterval(6),
        ),
        item_class=pitchtools.NumberedInterval,
      ),
      ratio=mathtools.Ratio((1, 1)),
    ),
    phrase_inflections=consort.tools.RegisterInflectionInventory(
      [      consort.tools.RegisterInflection(
        inflections=pitchtools.IntervalSegment(
          (      pitchtools.NumberedInterval(3),
            pitchtools.NumberedInterval(-3),
          ),
          item_class=pitchtools.NumberedInterval,
        ),
        ratio=mathtools.Ratio((1,)),
      ),
      segment_inflections=consort.tools.RegisterInflectionInventory(
        [      consort.tools.RegisterInflection(
          inflections=pitchtools.IntervalSegment(
            (        )
        ),
      ]
    ]
  ]
)
inflections=pitchtools.IntervalSegment(
    (pitchtools.NumberedInterval(-12),
     pitchtools.NumberedInterval(-9),
     pitchtools.NumberedInterval(0),
     pitchtools.NumberedInterval(12),
    ),
    item_class=pitchtools.NumberedInterval,
    ),
    ratio=mathtools.Ratio((3, 2, 1)),
    )

base_pitch = self.base_pitch or pitchtools.NamedPitch('C4')
base_pitch = base_pitch.transpose(transposition)
return new(self, base_pitch=base_pitch)

### PUBLIC PROPERTIES ###

@property
def base_pitch(self):
    return self._base_pitch
@property
def division_inflections(self):
    return self._division_inflections
@property
def phrase_inflections(self):
    return self._phrase_inflections
@property
def segment_inflections(self):
    return self._segment_inflections

A.42  consort.tools.RegisterSpecifierInventory

class RegisterSpecifierInventory(datastructuretools.TypedList):
    __slots__ = ()

    @property
def _attribute_manifest(self):

# -*- encoding: utf-8 -*-
from abjad.tools import datastructuretools
class RegisterSpecifierInventory(datastructuretools.TypedList):
    __slots__ = ()

    @property
def _attribute_manifest(self):
from abjad.tools import systemtools
return systemtools.AttributeManifest()

@property
def _item_callable(self):
    import consort
    return consort.RegisterSpecifier

A.43 consort.tools.ScoreTemplate

# -*- encoding: utf-8 -*-
import abc
from abjad import attach
from abjad.tools import abctools
from abjad.tools import indicatortools
from abjad.tools import instrumenttools
from abjad.tools import scoretools
from abjad.tools import stringtools

class ScoreTemplate(abctools.AbjadValueObject):
    ### CLASS VARIABLES ###
    __slots__ =
        '_composite_context_pairs',
        '_context_name_abbreviations',
    _is_populated = False
    ### INITIALIZER ###
def __init__(self):
    self._context_name_abbreviations = {}
    self._composite_context_pairs = {}
    if not type(self)._is_populated:
        self()  
        type(self)._is_populated = True
    ### SPECIAL METHODS ###
@abc.abstractmethod
def __call__(self):
    raise NotImplementedError
    ### PRIVATE METHODS ###
def _attach_tag(self, label, context):
    label = stringtools.to_dash_case(label)
    tag = indicatortools.LilyPondCommand(
        name="tag #'{}".format(label),
        format_slot='before',
        )
def _make_voice(self, name, abbreviation=None, context_name=None):
    name = name.title()
    abbreviation = abbreviation or name
    abbreviation = stringtools.to_snake_case(abbreviation)
    voice_name = '{} Voice'.format(name)
    voice = scoretools.Voice(
        name=voice_name,
        context_name=context_name,
    )
    self._context_name_abbreviations[abbreviation] = voice.name
    return voice

def _make_staff(
    self,
    name,
    clef,
    abbreviation=None,
    context_name=None,
    instrument=None,
    tag=None,
):
    name = name.title()
    staff_name = '{} Staff'.format(name)
    context_name = context_name or staff_name
    abbreviation = abbreviation or name
    abbreviation = stringtools.to_snake_case(abbreviation)
    voice = self._make_voice(name, abbreviation=abbreviation)
    staff = scoretools.Staff(
        [voice],
        context_name=context_name,
        name=staff_name
    )
    if not isinstance(clef, indicatortools.Clef):
        clef = indicatortools.Clef(clef)
    attach(clef, staff)
    if tag:
        self._attach_tag(tag, staff)
    if instrument:
        assert isinstance(instrument, instrumenttools.Instrument)
        attach(instrument, staff)
    return staff

def _populate(self):
    if not type(self)._is_populated:
        self()
        type(self)._is_populated = True

### PUBLIC PROPERTIES ###
@property
def all_voice_names(self):
    self._populate()
    result = []
    for name in self.context_name_abbreviations:
        if name in self.composite_context_pairs:
            continue
        result.append(name)
    return tuple(result)

@property
def composite_context_pairs(self):
    self._populate()
    return self._composite_context_pairs

@property
def context_name_abbreviations(self):
    self._populate()
    return self._context_name_abbreviations

A.44  consort.tools.ScoreTemplateManager

# -*- encoding: utf-8 -*-
from abjad import attach
from abjad import set_
from abjad.tools import abctools
from abjad.tools import indicatortools
from abjad.tools import markuptools
from abjad.tools import scoretools
from abjad.tools import stringtools

class ScoreTemplateManager(abctools.AbjadObject):

    ### PUBLIC METHODS ###

    @staticmethod
    def attach_tag(label, context):
        label = stringtools.to_dash_case(label)
        tag = indicatortools.LilyPondCommand(
            name="tag #('{}').format(label),
            format_slot='before',
        )
        attach(tag, context)

    @staticmethod
    def make_auxiliary_staff(
        primary_instrument=None,
        secondary_instrument=None,
        score_template=None,
    ):
        name = '{} {}'.format(
            primary_instrument.instrument_name.title(),
            secondary_instrument.instrument_name.title(),
        )
voice = scoretools.Voice(
    name='{} Voice'.format(name),
)
context_name = ScoreTemplateManager.make_staff_name(
    secondary_instrument.instrument_name.title(),
)
staff = scoretools.Staff([voice],
    name='{} Staff'.format(name),
    context_name=context_name,
)
abbreviation = stringtools.to_snake_case(name)
score_template._context_name_abbreviations[abbreviation] = voice.name
return staff

@staticmethod
def make_column_markup(string, space):
    string_parts = string.split()
    if len(string_parts) == 1:
        markup = markuptools.Markup(string_parts[0]).hcenter_in(space)
    else:
        markups = [markuptools.Markup(_).hcenter_in(space) for _ in string_parts]
        markup = markuptools.center_column(markups, direction=None)
    markup = markup.hcenter_in(space)
    return markup

@staticmethod
def make_ensemble_group(label=None, name=None, performer_groups=None):,
    ensemble_group = scoretools.StaffGroup(
        performer_groups,
        name=name,
        context_name='EnsembleGroup',
    )
    if label is not None:
        ScoreTemplateManager.attach_tag(label, ensemble_group)
    return ensemble_group

@staticmethod
def make_performer_group(
    context_name=None, instrument=None, label=None,
):
    context_name = context_name or 'PerformerGroup'
    name = '{} Performer Group'.format(instrument.instrument_name.title())
    performer_group = scoretools.StaffGroup(
        context_name=context_name,
        name=name,
    )
    performer_group.is_simultaneous = True
if label is not None:
    ScoreTemplateManager.attach_tag(label, performer_group)
attach(
    instrument,
    performer_group,
    scope=context_name,
    is_annotation=True,
)
manager = set_(performer_group)
manager.instrument_name = instrument.instrument_name_markup
manager.short_instrument_name = instrument.short_instrument_name_markup
return performer_group

@staticmethod
def make_single_basic_performer(
    clef=None,
    context_name=None,
    instrument=None,
    label=None,
    score_template=None,
):
    performer_group = ScoreTemplateManager.make_performer_group(
        context_name=context_name,
        instrument=instrument,
        label=label,
    )
    name = instrument.instrument_name.title()
    context_name = ScoreTemplateManager.make_staff_name(name)
    voice = scoretools.Voice(
        name='{} Voice'.format(name),
    )
    staff = scoretools.Staff(
        [voice],
        context_name=context_name,
        name='{} Staff'.format(name),
    )
    performer_group.append(staff)
    attach(clef, voice)
    abbreviation = stringtools.to_snake_case(name)
    score_template._context_name_abbreviations[abbreviation] = voice.name
    return performer_group

@staticmethod
def make_single_piano_performer(
    instrument=None,
    score_template=None,
):
    performer_group = ScoreTemplateManager.make_performer_group(
        context_name='PianoStaff',
        instrument=instrument,
        label=stringtools.to_dash_case(instrument.instrument_name),
    )
    name = instrument.instrument_name.title()
    upper_voice = scoretools.Voice(
name='{} Upper Voice'.format(name),
)
upper_staff = scoretools.Staff(
    [upper_voice],
    context_name='PianoUpperStaff',
    name='{} Upper Staff'.format(name),
)
dynamics = scoretools.Voice(
    context_name='Dynamics',
    name='{} Dynamics'.format(name),
)
lower_voice = scoretools.Voice(
    name='{} Lower Voice'.format(name),
)
lower_staff = scoretools.Staff(
    [lower_voice],
    context_name='PianoLowerStaff',
    name='{} Lower Staff'.format(name),
)
pedals = scoretools.Voice(
    context_name='Dynamics',
    name='{} Pedals'.format(name),
)
performer_group.extend((
    upper_staff,
    dynamics,
    lower_staff,
    pedals,
))
attach(indicatortools.Clef('treble'), upper_voice)
attach(indicatortools.Clef('bass'), lower_voice)
score_template._context_name_abbreviations.update(
    piano_dynamics=dynamics.name,
    piano_lh=lower_voice.name,
    piano_pedals=pedals.name,
    piano_rh=upper_voice.name,
)
return performer_group

@staticmethod
def make_single_string_performer(
    abbreviation=None,
    clef=None,
    instrument=None,
    score_template=None,
    split=True,
):
    performer_group = ScoreTemplateManager.make_performer_group(
        context_name='StringPerformerGroup',
        instrument=instrument,
        label=stringtools.to_dash_case(instrument.instrument_name),
    )
    name = instrument.instrument_name.title()
    abbreviation = abbreviation or \
stringtools.to_snake_case(name)

if split:
    right_hand_voice = scoretools.Voice(
        name='{} Bowing Voice'.format(name),
    )
    right_hand_staff = scoretools.Staff(
        [right_hand_voice],
        context_name='BowingStaff',
        name='{} Bowing Staff'.format(name),
    )
    left_hand_voice = scoretools.Voice(
        name='{} Fingering Voice'.format(name),
    )
    left_hand_staff = scoretools.Staff(
        [left_hand_voice],
        context_name='FingeringStaff',
        name='{} Fingering Staff'.format(name),
    )
    performer_group.append(right_hand_staff)
    performer_group.append(left_hand_staff)
    attach(clef, left_hand_staff)
    attach(indicatortools.Clef('percussion'), right_hand_staff)
    right_hand_abbreviation = '{}_rh'.format(abbreviation)
    left_hand_abbreviation = '{}_lh'.format(abbreviation)
    score_template._context_name_abbreviations.update({
        abbreviation: performer_group.name,
        right_hand_abbreviation: right_hand_voice.name,
        left_hand_abbreviation: left_hand_voice.name,
    })
    score_template._composite_context_pairs[abbreviation] = (
        right_hand_abbreviation,
        left_hand_abbreviation,
    )
else:
    voice = scoretools.Voice(
        name='{} Voice'.format(name),
    )
    staff = scoretools.Staff(
        [voice],
        context_name='StringStaff',
        name='{} Staff'.format(name),
    )
    performer_group.append(staff)
    attach(clef, voice)
    score_template._context_name_abbreviations[abbreviation] = \
        voice.name
    return performer_group
@staticmethod
def make_single_wind_performer(
    abbreviation=None,
    clef=None,
    instrument=None,
    score_template=None,

A.45  consort.tools.SeedSession

```python
# -*- encoding: utf-8 -*-

from abjad import abctools

class SeedSession(abctools.AbjadObject):
    
    ### CLASS VARIABLES ###
    
    @staticmethod
    def make_staff_name(name):
        name = ''.join(x for x in name if x.isalpha())
        name = '{}Staff'.format(name)
        return name

    @staticmethod
    def make_voice_name(name):
        name = ''.join(x for x in name if x.isalpha())
        name = '{}Voice'.format(name)
        return name

    @staticmethod
    def make_time_signature_context():
        time_signature_context = scoretools.Context(
            context_name='TimeSignatureContext',
            name='Time Signature Context',
            )
        label = 'time'
        ScoreTemplateManager.attach_tag(label, time_signature_context)
        return time_signature_context

    performer_group = ScoreTemplateManager.make_performer_group(
        instrument=instrument,
        label=stringtools.to_dash_case(instrument.instrument_name),
        )
    name = instrument.instrument_name.title()
    context_name = ScoreTemplateManager.make_staff_name(name)
    voice = scoretools.Voice(
        name='{} Voice'.format(name),
        )
    staff = scoretools.Staff(
        [voice],
        context_name=context_name,
        name='{} Staff'.format(name),
        )
    performer_group.append(staff)
    attach(clef, voice)
    abbreviation = abbreviation or \ 
        stringtools.to_snake_case(name)
    score_template._context_name_abbreviations[abbreviation] = voice.name
    return performer_group
```
__slots__ = (
    '_current_timewise_logical_tie_seed',
    '_current_timewise_phrase_seed',
    '_current_phrased_voicewise_logical_tie_seed',
    '_current_unphrased_voicewise_logical_tie_seed',
    '_current_timewise_musicSpecifier_seed',
    '_timewise_logical_tie_seeds',
    '_timewise_musicSpecifier_seeds',
    '_timewise_phrase_seeds',
    '_phrased_voicewise_logical_tie_seeds',
    '_unphrased_voicewise_logical_tie_seeds',
)

### INITIALIZER ###

def __init__(self):
    self._current_timewise_logical_tie_seed = 0
    self._current_timewise_phrase_seed = 0
    self._current_phrased_voicewise_logical_tie_seed = 0
    self._current_timewise_musicSpecifier_seed = 0
    self._timewise_musicSpecifier_seeds = {}
    self._timewise_logical_tie_seeds = {}
    self._timewise_phrase_seeds = {}
    self._phrased_voicewise_logical_tie_seeds = {}
    self._unphrased_voicewise_logical_tie_seeds = {}

### SPECIAL METHODS ###

def __call__(self,
             application_rate,
             attack_point_signature,
             music_specifier,
             voice,
             ):
    phrased_voicewise_logical_tie_seed = self._get_phrased_voicewise_logical_tie_seed(
        attack_point_signature,
        music_specifier,
        application_rate,
        voice,
    )
    unphrased_voicewise_logical_tie_seed = self._get_unphrased_voicewise_logical_tie_seed(
        music_specifier,
        voice,
    )
    timewise_phrase_seed = self._get_timewise_phrase_seed(
        attack_point_signature,
        music_specifier,
        voice,
    )
    timewise_musicSpecifier_seed = \
        self._get_timewise_musicSpecifier_seed(
            music_specifier,
        )
```python
self._current_timewise_phrase_seed = timewise_phrase_seed
self._current_phrased_voicewise_logical_tie_seed = \n    phrased_voicewise_logical_tie_seed
self._current_unphrased_voicewise_logical_tie_seed = \n    unphrased_voicewise_logical_tie_seed
self._current_timewise_musicSpecifier_seed = \n    timewise_musicSpecifier_seed

### PRIVATE METHODS ###

def _get_timewise_musicSpecifier_seed(
    self,
    musicSpecifier,
):  
    if musicSpecifier not in self._timewise_musicSpecifier_seeds:
        self._timewise_musicSpecifier_seeds[musicSpecifier] = 0
        seed = self._timewise_musicSpecifier_seeds[musicSpecifier]
        self._timewise_musicSpecifier_seeds[musicSpecifier] += 1
    return seed

def _get_timewise_phrase_seed(
    self,
    attack_pointSignature,
    musicSpecifier,
    voice,
):  
    key = (voice, musicSpecifier)
    if attack_pointSignature.is_first_of_phrase:
        if key not in self._timewise_phrase_seeds:
            phrase_seed = (musicSpecifier.seed or 0) - 1
            self._timewise_phrase_seeds[key] = phrase_seed
            self._timewise_phrase_seeds[key] += 1
        phrase_seed = self._timewise_phrase_seeds[key]
    return phrase_seed

def _get_phrased_voicewise_logical_tie_seed(
    self,
    attack_pointSignature,
    musicSpecifier,
    applicationRate,
    voice,
):  
    if musicSpecifier not in self._timewise_logical_tie_seeds:
        seed = (musicSpecifier.seed or 0) - 1
        self._timewise_logical_tie_seeds[musicSpecifier] = seed
        self._phrased_voicewise_logical_tie_seeds[voice] = seed
    if applicationRate == 'phrase':
        if attack_pointSignature.is_first_of_phrase:
            self._timewise_logical_tie_seeds[musicSpecifier] += 1
            seed = self._timewise_logical_tie_seeds[musicSpecifier]
            self._phrased_voicewise_logical_tie_seeds[voice] = seed
        else:
            seed = self._phrased_voicewise_logical_tie_seeds[voice]
    elif applicationRate == 'division':
```

611
if attack_point_signature.is_first_of_division:
    self._timewise_logical_tie_seeds[music_specifier] += 1
    seed = self._timewise_logical_tie_seeds[music_specifier]
    self.phrased_voicewise_logical_tie_seeds[voice] = seed
else:
    seed = self.phrased_voicewise_logical_tie_seeds[voice]
else:
    self._timewise_logical_tie_seeds[music_specifier] += 1
    seed = self._timewise_logical_tie_seeds[music_specifier]
return seed

def _get_unphrased_voicewise_logical_tie_seed(self, music_specifier, voice):
    if voice not in self.unphrased_voicewise_logical_tie_seeds:
        self.unphrased_voicewise_logical_tie_seeds[voice] = {}
    if music_specifier not in self.unphrased_voicewise_logical_tie_seeds[voice]:
        self.unphrased_voicewise_logical_tie_seeds[voice][music_specifier] = 
        (music_specifier.seed or 0) - 1
    self.unphrased_voicewise_logical_tie_seeds[voice][music_specifier] += 1
    return self.unphrased_voicewise_logical_tie_seeds[voice][music_specifier]

### PUBLIC PROPERTIES ###

@property
def current_timewise_musicSpecifier_seed(self):
    return self._current_timewise_music_specifier_seed

@property
def current_timewise_phrase_seed(self):
    return self._current_timewise_phrase_seed

@property
def current_phrased_voicewise_logical_tie_seed(self):
    return self._current_phrased_voicewise_logical_tie_seed

@property
def current_unphrased_voicewise_logical_tie_seed(self):
    return self._current_unphrased_voicewise_logical_tie_seed

A.46 consort.tools.SegmentMaker

# -*- encoding: utf-8 -*-
from __future__ import print_function
import collections
import importlib
import itertools
import os
from abjad import attach
from abjad import detach
from abjad import inspect_
from abjad import iterate

612
class SegmentMaker(makertools.SegmentMaker):
    r'''A Consort segment-maker.

    ::

        >>> import consort
        >>> score_template = templatetools.StringOrchestraScoreTemplate(
        ...     violin_count=2,
        ...     viola_count=1,
        ...     cello_count=1,
        ...     contrabass_count=0,
        ... )
        >>>

    ::

        >>> segment_maker = consort.SegmentMaker(
        ...     score_template=score_template,
        ...     settings=(
        ...         consort.MusicSetting(
        ...             timespan_maker=consort.TaleaTimespanMaker(),
        ...             violin_1_bowing_voice=consort.MusicSpecifier(),
        ...             violin_2_bowing_voice=consort.MusicSpecifier(),
        ...         ),
        ...         desired_duration_in_seconds=2,
        ...         tempo=indicatortools.Tempo((1, 4), 72),
        ...         permitted_time_signatures=(
        ...             (5, 8),
        ...             (7, 16),
        ...         ),
        ...     ),
        ... )
        >>> print(format(segment_maker))

        consort.tools.SegmentMaker(

        )
desired_duration_in_seconds=durationtools.Duration(2, 1),
permitted_time_signatures=indicatortools.TimeSignatureInventory(
    [indicatortools.TimeSignature((5, 8)), indicatortools.TimeSignature((7, 16))],
score_template=templatetools.StringOrchestraScoreTemplate(
    violin_count=2,
    viola_count=1,
    cello_count=1,
    contrabass_count=0,
    split_hands=True,
    use_percussion_clefs=False,
),
settings=(
    consort.tools.MusicSetting(
        timespan_maker=consort.tools.TaleaTimespanMaker(
            playing_talea=rhythmmakertools.Talea(
                counts=(4,),
                denominator=16,
            ),
            playing_groupings=(1,),
            repeat=True,
            silence_talea=rhythmmakertools.Talea(
                counts=(4,),
                denominator=16,
            ),
            step_anchor=Right,
            synchronize_groupings=False,
            synchronize_step=False,
        ),
        violin_1_bowing_voice=consort.tools.MusicSpecifier(),
        violin_2_bowing_voice=consort.tools.MusicSpecifier(),
    ),
    tempo=indicatortools.Tempo(
        duration=durationtools.Duration(1, 4),
        units_per_minute=72,
    ),
)
::

>>> lilypond_file = segment_maker() # doctest: +SKIP

Performing rhythmic interpretation:
    populating independent timespans:
    populated timespans: ...
    found meters: ...
    demultiplexed timespans: ...
    split timespans: ...
    pruned malformed timespans: ...
    consolidated timespans: ...
    inscribed timespans: ...
multiplexed timespans: ...
pruned short timespans: ...
pruned meters: ...
total: ...
populating dependent timespans:
populated timespans: ...
demultiplexed timespans: ...
split timespans: ...
pruned short timespans: ...
pruned malformed timespans: ...
consolidated timespans: ...
inscribed timespans: ...
total: ...
populated silent timespans: ...
validated timespans: ...
rewriting meters:
  rewriting Cello Bowing Voice: 2
  rewriting Cello Fingering Voice: 2
  rewriting Viola Bowing Voice: 2
  rewriting Viola Fingering Voice: 2
  rewriting Violin 1 Bowing Voice: 3
  rewriting Violin 1 Fingering Voice: 2
  rewriting Violin 2 Bowing Voice: 3
  rewriting Violin 2 Fingering Voice: 2
  total: 0.16949145279
populated score: ...
total: ...
Performing non-rhythmic interpretation:
collected attack points: ...
handled graces: ...
handled pitches: ...
handled attachments: ...
total: ...
Checking for well-formedness violations:
  [] 24 check_beamed_quarter_notes
  [] 18 check_discontiguous_spanners
  [] 80 check_duplicate_ids
  [] 0 check_intermarked_hairpins
  [] 2 check_misdurated_measures
  [] 2 check_misfilled_measures
  [] 4 check_mispitched_ties
  [] 24 check_misrepresented_flags
  [] 80 check_missing_parents
  [] 2 check_nested_measures
  [] 0 check_overlapping_beams
  [] 0 check_overlapping_glissandi
  [] 0 check_overlapping_octavation_spanners
  [] 0 check_short_hairpins
  total: ...

'

### CLASS VARIABLES ###

```python
615
```
```python
__slots__ = (
    '_annotate_colors',
    '_annotate_phrasing',
    '_annotate_timespans',
    '_attack_point_map',
    '_desired_duration_in_seconds',
    '_discard_final_silence',
    '_lilypond_file',
    '_maximum_meter_run_length',
    '_meters',
    '_name',
    '_omit_stylesheets',
    '_permitted_time_signatures',
    '_previous_segment_metadata',
    '_repeat',
    '_score',
    '_score_template',
    '_segment_metadata',
    '_settings',
    '_tempo',
    '_timespan_quantization',
    '_voice_names',
    '_voicewise_timespans',
)

### INITIALIZER ###

def __init__(self,
    annotate_colors=None,
    annotate_phrasing=None,
    annotate_timespans=None,
    desired_duration_in_seconds=None,
    discard_final_silence=None,
    maximum_meter_run_length=None,
    name=None,
    omit_stylesheets=None,
    permitted_time_signatures=None,
    repeat=None,
    score_template=None,
    settings=None,
    tempo=None,
    timespan_quantization=None,
):
    makertools.SegmentMaker.__init__(self,
        self.name = name
        self.annotate_colors = annotate_colors
        self.annotate_phrasing = annotate_phrasing
        self.annotate_timespans = annotate_timespans
        self.desired_duration_in_seconds = desired_duration_in_seconds
        self.discard_final_silence = discard_final_silence
        self.maximum_meter_run_length = maximum_meter_run_length
```
```python
self.omit_stylesheets = omit_stylesheets
self.permitted_time_signatures = permitted_time_signatures
self.score_template = score_template
self.tempo = tempo
self.timespan_quantization = timespan_quantization
self.settings = settings
self.repeat = repeat
self._reset()

### SPECIAL METHODS ###

def __call__(
    self,
    annotate=None,
    verbose=True,
    segment_metadata=None,
    previous_segment_metadata=None,
):
    import consort

    self._reset()

    self._annotate_phrasing = self._annotate_phrasing or annotate
    self._segment_metadata = segment_metadata or \
        datastructuretools.TypedOrderedDict()
    self._previous_segment_metadata = previous_segment_metadata or \
        datastructuretools.TypedOrderedDict()
    self._score = self.score_template()
    self._voice_names = tuple(
        voice.name for voice in
        iterate(self.score).by_class(scoretools.Voice)
    )

    with systemtools.Timer(
        total='Performing rhythmic interpretation:',
        verbose=verbose,
    ):  
        self.interpret_rhythms(verbose=verbose)
    with systemtools.Timer(
        total='Performing non-rhythmic interpretation:',
        verbose=verbose,
    ):  
        with systemtools.Timer(
            total='collected attack points:',
            verbose=verbose,
        ):  
            attack_point_map = self.collect_attack_points(self.score)
    self._attack_point_map = attack_point_map
    with systemtools.ForbidUpdate(self.score, update_on_exit=True):
        with systemtools.Timer(
            total='handled graces:',
            verbose=verbose,
        ):  
            ...  
```

```
consort.GraceHandler._process_session(self)
with systemtools.ForbidUpdate(self.score, update_on_exit=True):
    with systemtools.Timer(
        ' total:',
        ' handling pitches:',
        verbose=verbose,
    ):
        consort.PitchHandler._process_session(self)
with systemtools.ForbidUpdate(self.score, update_on_exit=True):
    with systemtools.Timer(
        ' total:',
        ' handling attachments:',
        verbose=verbose,
    ):
        consort.AttachmentHandler._process_session(
            self,
            verbose=verbose,
        )
    self.configure_score()
    self.configure_lilypond_file()
with systemtools.Timer(
    enter_message='Checking for well-formedness violations:',
    exit_message=' total:',
    verbose=verbose,
):
    self.validate_score(self.score, verbose=verbose)
    self.update_segment_metadata()

return self.lilypond_file, self._segment_metadata

### PRIVATE METHODS ###

def _reset(self):
    self._attack_point_map = None
    self._lilypond_file = None
    self._meters = None
    self._score = None
    self._voice_names = None
    self._voicewise_timespans = None
    self._segment_metadata = None
    self._previous_segment_metadata = None

### PRIVATE PROPERTIES ###

@property
def _storage_format_specification(self):
    from abjad.tools import systemtools
    manager = systemtools.StorageFormatManager
    keyword_argument_names = manager.get_keyword_argument_names(self)
    keyword_argument_names = list(keyword_argument_names)
    if not self.settings:
        keyword_argument_names.remove('settings')
    return systemtools.StorageFormatSpecification(
```

```python
### PUBLIC METHODS ###

def get_end_instruments(self):
    result = datastructuretools.TypedOrderedDict()
    staves = iterate(self._score).by_class(scoretools.Staff)
    staves = list(staves)
    staves.sort(key=lambda x: x.name)
    prototype = (instrumenttools.Instrument,)
    for staff in staves:
        last_leaf = inspect_(staff).get_leaf(-1)
        instrument = inspect_(last_leaf).get_effective(prototype)
        if instrument:
            result[staff.name] = instrument.instrument_name
        else:
            result[staff.name] = None
    return result

def get_end_tempo_indication(self):
    prototype = indicatortools.Tempo
    context = self._score['Time Signature Context']
    last_leaf = inspect_(context).get_leaf(-1)
    effective_tempo = inspect_(last_leaf).get_effective(prototype)
    if effective_tempo is not None:
        duration = effective_tempo.duration.pair
        units_per_minute = effective_tempo.units_per_minute
        effective_tempo = (duration, units_per_minute)
    return effective_tempo

def get_end_time_signature(self):
    prototype = indicatortools.TimeSignature
    context = self._score['Time Signature Context']
    last_measure = context[-1]
    time_signature = inspect_(last_measure).get_effective(prototype)
    if not time_signature:
        return
    pair = time_signature.pair
    return pair

def add_time_signature_context(self):
    import consort
    if 'Time Signature Context' not in self.score:
        time_signature_context = \
            consort.ScoreTemplateManager.make_time_signature_context()
        self.score.insert(0, time_signature_context)
    context = self.score['Time Signature Context']
    time_signatures = [_.implied_time_signature for _ in self.meters]
    iterator = itertools.groupby(time_signatures, lambda x: x)
    measures = []
    for time_signature, group in iterator:
        count = len(tuple(group))
```

skip = scoretools.Skip()
multiplier = durationtools.Multiplier(time_signature) * count
attach(multiplier, skip)
attach(time_signature, skip, scope=scoretools.Score)
measure = scoretools.Container([skip])
measures.append(measure)
context.extend(measures)
def add_setting(self, silenced_contexts=None, timespan_identifier=None, timespan_maker=None, **music_specifiers):
    import consort
    setting = consort.MusicSetting(silenced_contexts=silenced_contexts, timespan_identifier=timespan_identifier, timespan_maker=timespan_maker, **music_specifiers)
    self._settings.append(setting)
def attach_initial_bar_line(self):
    segment_number = self._segment_metadata.get('segment_number', 1)
    if self.repeat:
        if segment_number != 1:
            command = indicatortools.LilyPondCommand('break', 'opening')
            attach(command, self.score['Time Signature Context'])
            return
        elif self._previous_segment_metadata.get('is_repeated'):
            return
        elif segment_number == 1:
            bar_line = indicatortools.LilyPondCommand('bar "||"', 'opening')
            for staff in iterate(self.score).by_class(scoretools.Staff):
                attach(bar_line, staff)

def attach_final_bar_line(self):
    segment_number = self._segment_metadata.get('segment_number', 1)
    segment_count = self._segment_metadata.get('segment_count', 1)
    if self.repeat:
        repeat = indicatortools.Repeat()
        for staff in iterate(self.score).by_class(scoretools.Staff):
            attach(repeat, staff)
        attach(repeat, self.score['Time Signature Context'])
    elif segment_number == segment_count:
        self.score.add_final_bar_line(
            abbreviation='|.',
            to_each_voice=True,
        )
    if segment_number == segment_count and self.final_markup is not None:
        self.score.add_final_markup(self.final_markup)
def get_rehearsal_letter(self):
    segment_number = self._segment_metadata.get('segment_number', 1)
    if segment_number == 1:
        return ''
    segment_index = segment_number - 1
    rehearsal_ordinal = ord('A') - 1 + segment_index
    rehearsal_letter = chr(rehearsal_ordinal)
    return rehearsal_letter

def attach_rehearsal_mark(self):
    markup_a, markup_b = None, None
    first_leaf = self.score['Time Signature Context'].select_leaves()[0]
    rehearsal_letter = self.get_rehearsal_letter()
    if rehearsal_letter:
        markup_a = markuptools.Markup(rehearsal_letter)
        markup_a = markup_a.caps().pad_around(0.5).box()
        if self.name:
            markup_b = markuptools.Markup('"{}"'.format(self.name or ''))
            markup_b = markup_b.fontsize(-3)
        if markup_a and markup_b:
            markup = markuptools.Markup.concat([markup_a, ' ', markup_b])
        else:
            markup = markup_a or markup_b
    if markup:
        rehearsal_mark = indicatortools.RehearsalMark(markup=markup)
        attach(rehearsal_mark, first_leaf)

def attach_tempo(self):
    first_leaf = self.score['Time Signature Context'].select_leaves()[0]
    if self.tempo is not None:
        attach(self.tempo, first_leaf)

def configure_lilypond_file(self):
    lilypond_file = lilypondfiletools.LilyPondFile()
    if not self.omitStylesheets:
        lilypond_file.use_relative_includes = True
        path = os.path.join(
            '..', '', '
            'stylesheets', 
            'stylesheet.ily',
        )
        lilypond_file.file_initial_user_includes.append(path)
        if 1 < self._segment_metadata.get('segment_number', 1):
            path = os.path.join(
                '..', '', 
                'stylesheets', 
                'nonfirst-segment.ily',
            )
            lilypond_file.file_initial_user_includes.append(path)
    lilypond_file.file_initial_system_comments[:] = []
    score_block = lilypondfiletools.Block(name='score')
score_block.items.append(self.score)
lilypond_file.items.append(score_block)
lilypond_file.score = self.score
self._lilypond_file = lilypond_file

def configure_score(self):
    self.add_time_signature_context()
    self.attach_tempo()
    self.attach_rehearsal_mark()
    self.attach_initial_bar_line()
    self.attach_final_bar_line()
    self.set_bar_number()
    self.postprocess_grace_containers()
    self.postprocess_ties()
    self.postprocess_staff_lines_spanners()
    self.postprocess_multimeasure_rests()
    self.attach_bar_number_comments()
    self.apply_annotations()

def apply_annotations(self):
    import consort
    if self.annotate_phrasing:
        consort.annotate(self.score, nonsilence=True)
    if self.annotate_timespans:
        context = self.score['Time Signature Context']
        for leaf in iterate(context).by_class(scoretools.Leaf):
            timespan = inspect_(leaf).get_timespan()
            start_fraction = markuptools.Markup.fraction(
                timespan.start_offset)
            stop_fraction = markuptools.Markup.fraction(
                timespan.stop_offset)
            markup_contents = [start_fraction, ' : ', stop_fraction]
            markup = markuptools.Markup.concat(markup_contents)
            markup = markuptools.Markup(markup, Up)
            markup = markup.pad_around(0.5).box()
            attach(markup, leaf)
    if self.annotate_colors:
        for voice in iterate(self.score).by_class(scoretools.Voice):
            for phrase in voice:
                musicSpecifier = inspect_(phrase).get_indicator(
                    consort.MusicSpecifier)
                if musicSpecifier is None:
                    continue
                color = musicSpecifier.color
                if color is None:
                    continue
                override(phrase).beam.color = color
                override(phrase).dots.color = color
                override(phrase).flag.color = color
                override(phrase).note_head.color = color
                override(phrase).stem.color = color

def postprocess_multimeasure_rests(self):
    def division_to_meter(self, division):

offset = inspect_(division).get_timespan().start_offset
timespan = meter_timespans.find_timespans_starting_at(offset)[0]
meter = timespan.annotation
return meter

import consort
silentSpecifier = consort.MusicSpecifier()
meter_timespans = self.meters_to_timespans(self.meters)
with systemtools.ForbidUpdate(self.score):
    for voice in iterate(self.score).by_class(scoretools.Voice):
        for phrase in voice:
            music_specifier = inspect_(phrase).get_indicator(
                consort.MusicSpecifier)
            if music_specifier != silentSpecifier:
                continue
divisions = [_.for_._in phrase
                if isinstance(_.[0], scoretools.MultimeasureRest)
                ]
        iterator = itertools.groupby(divisions, division_to_meter)
        for meter, grouped_divisions in iterator:
            grouped_divisions = list(grouped_divisions)
            count = len(grouped_divisions)
            if count == 1:
                continue
            for division in grouped_divisions[1:]:
                phrase.remove(division)
                rest = grouped_divisions[0][0]
                multiplier = inspect_(rest).get_indicator(
                    durationtools.Multiplier)
                detach(multiplier, rest)
                multiplier = multiplier * count
                attach(multiplier, rest)

def postprocess_staff_lines_spanners(self):
    segment_number = self._segment_metadata.get('segment_number', 1)
    segment_count = self._segment_metadata.get('segment_count', 1)
    if segment_number != segment_count:
        return
    for voice in iterate(self.score).by_class(scoretools.Voice):
        for leaf in iterate(voice).by_class(scoretools.Leaf, reverse=True):
            if not isinstance(leaf, scoretools.MultimeasureRest):
                break
            prototype = spannertools.StaffLinesSpanner
            if not inspect_(leaf).has_spanner(prototype):
                continue
            staff_lines_spanner = inspect_(leaf).get_spanner(prototype)
            components = staff_lines_spanner.components
            detach(staff_lines_spanner)
            staff_lines_spanner = new(
                staff_lines_spanner,
                forbid_restarting=True,
            )
            attach(
                staff_lines_spanner,
                components,
def attach_bar_number_comments(self):
    first_bar_number = self._segment_metadata.get('first_bar_number', 1)
    measure_offsets = self.measure_offsets
    for voice in iterate(self.score).by_class(scoretools.Voice):
        voice_name = voice.name
        for phrase in voice:
            for division in phrase:
                timespan = inspect_(division).get_timespan()
                start_offset = timespan.start_offset
                matched = False
                for bar_number, measure_offset in enumerate(measure_offsets, first_bar_number):
                    if measure_offset == start_offset:
                        matched = True
                        break
                if not matched:
                    continue
                string = '[{} Measure {}].format(
                    voice_name,
                    bar_number,
                )
                comment = indicatortools.LilyPondComment(string)
                attach(comment, division)

def postprocess_ties(self):
    for component in iterate(self.score).depth_first():
        if not inspect_(component).has_spanner(spannertools.Tie):
            continue
        tie = inspect_(component).get_spanner(spannertools.Tie)
        if component != tie[0]:
            continue
        components = tie.components
        detach(tie)
        tie = spannertools.Tie(use_messiaen_style_ties=True)
        attach(tie, components)

def set_bar_number(self):
    first_bar_number = self._segment_metadata.get('first_bar_number')
    if first_bar_number is not None:
        set_(self.score).current_bar_number = first_bar_number
    else:
        # override(self.score).bar_number.transparent = True

def copy_voice(self, voice, attachment_names=None, new_voice_name=None, new_context_name=None, remove_grace_containers=False,
remove_ties=False,
replace_rests_with_skips=False,
):
    new_voice = mutate(voice).copy()
    if new_voice_name:
        new_voice.name = new_voice_name
    if new_context_name:
        new_voice.context_name = new_context_name
    rests = []
    for component in iterate(new_voice).depth_first(capped=True):
        agent = inspect_(component)
        indicators = agent.get_indicators(unwrap=False)
        spanners = agent.get_spanners()
        for x in indicators:
            if not x.name:
                continue
            if attachment_names and 
                not any(x.name.startswith(_) for _ in attachment_names):
                x._detach()
        for x in spanners:
            if remove_ties and isinstance(x, spannertools.Tie):
                x._detach()
            if not x.name:
                continue
            elif attachment_names and 
                not any(x.name.startswith(_) for _ in attachment_names):
                x._detach()
        if replace_rests_with_skips and 
            isinstance(component, scoretools.Rest):
            rests.append(component)
        grace_containers = agent.get_grace_containers()
        if grace_containers and remove_grace_containers:
            for grace_container in grace_containers:
                grace_container._detach()
        if replace_rests_with_skips:
            for rest in rests:
                indicators = inspect_(rest).get_indicators(  
                    durationtools.Multiplier,
                )
                skip = scoretools.Skip(rest)
                if indicators:
                    attach(indicators[0], skip)
                mutate(rest).replace(skip)
            return new_voice

@staticmethod
def logical_tie_to_musicSpecifier(logical_tie):
    import consort
    parentage = inspect_(logical_tie.head).get_parentage()
    musicSpecifier = None
    prototype = consort.MusicSpecifier
    for parent in parentage:
        if not inspect_(parent).has_indicator(prototype):
            continue
musicSpecifier = inspect_(parent).get_indicator(prototype)
return musicSpecifier

@staticmethod
def logical_tie_to_division(logical_tie):
    import consort
    parentage = inspect_(logical_tie.head).get_parentage()
    prototype = consort.MusicSpecifier
    for i, parent in enumerate(parentage):
        if inspect_(parent).has_indicator(prototype):
            break
    return parentage[i - 1]

@staticmethod
def logical_tie_to_phrase(logical_tie):
    import consort
    parentage = inspect_(logical_tie.head).get_parentage()
    prototype = consort.MusicSpecifier
    for parent in parentage:
        if inspect_(parent).has_indicator(prototype):
            return parent

@staticmethod
def logical_tie_to_voice(logical_tie):
    parentage = inspect_(logical_tie.head).get_parentage()
    voice = None
    for parent in parentage:
        if isinstance(parent, scoretools.Voice):
            voice = parent
            break
    return voice

@staticmethod
def logical_tie_to_staff(logical_tie):
    parentage = inspect_(logical_tie.head).get_parentage()
    staff = None
    for parent in parentage:
        if isinstance(parent, scoretools.Staff):
            staff = parent
            break
    return staff

def postprocess_grace_containers(self):
    import consort
    score = self.score
    stop_trill_span = consort.StopTrillSpan()
    for leaf in iterate(score).by_class(scoretools.Leaf):
        agent = inspect_(leaf)
        spanners = agent.get_spanners(consort.ConsortTrillSpanner)
        if not spanners:
            continue
        after_graces = agent.get_grace_containers('after')
        if not after_graces:
            continue
after_grace = after_graces[0]
leaf = after_grace[0]
attach(stop_trill_span, leaf)

@staticmethod
def validate_score(score, verbose=True):
    import consort
    manager = systemtools.WellformednessManager(expr=score)
    triples = manager()
    for current_violators, current_total, current_check in triples:
        if verbose:
            print('  {} {} {}
             current_violators,
            current_total,
            current_check,
            )
        if current_violators:
            raise AssertionError
        for voice in iterate(score).by_class(scoretools.Voice):
            #print(voice.name)
            voice_name = voice.name
            for phrase in voice:
                #print('PHRASE:', phrase)
                music_specifier = inspect_(phrase).get_indicator(
                    consort.MusicSpecifier)
                if music_specifier is None:
                    #print('	NO MUSIC SPECIFIER')
                    continue
                pitch_handler = music_specifier.pitch_handler
                if pitch_handler is not None:
                    if pitch_handler.pitches_are_nonsemantic:
                        #print('	PITCHES ARE NONSEMANTIC')
                        continue
                    instrument = music_specifier.instrument
                    if instrument is None:
                        instrument = inspect_(phrase).get_effective(
                            instrumenttools.Instrument)
                    if instrument is None:
                        #print('	NO INSTRUMENT')
                        continue
                    pitch_range = instrument.pitch_range
                    for leaf in iterate(phrase).by_class(
                        scoretools.Note, scoretools.Chord,
                        ):
                        timespan = inspect_(leaf).get_timespan()
                        #print('	 {!r}'.format(leaf))
                        if isinstance(leaf, scoretools.Note):
                            note_head = leaf.note_head
                            #print('\t\t', note_head)
                            if note_head_written_pitch not in pitch_range:
                                override(leaf).note_head.color = 'red'
                                message = ' \033[91m{}Out of range: {} {!r} {!s} {!s}{}'
                                message = message.format(\033[0m',

voice_name,
timespan,
pitch_range,
leaf,
'\033[0m',

print(message)

elif isinstance(leaf, scoretools.Chord):
    for note_head in leaf.note_heads:
        #print('\t\t', note_head)
        if note_head.written_pitch not in pitch_range:
            note_head.tweak.color = 'red'
            message = ' {}Out of range: {} {!r} {!s} {!s}{}'
            message = message.format(
                '\033[91m',
                voice_name,
timespan,
pitch_range,
leaf,
        note_head,
        '\033[0m',
)
            print(message)

@staticmethod
def can_rewrite_meter(inscribed_timespan):
    r'''Is true if containers to be inscribed into `inscribed_timespan` can
undergo meter rewriting. Otherwise false.

Returns boolean.
'''

import consort
musicSpecifier = inscribed_timespan.musicSpecifier
if musicSpecifier is None:
    return True
rhythm_maker = musicSpecifier.rhythm_maker
if rhythm_maker is None:
    return True
if isinstance(rhythm_maker, consort.CompositeRhythmMaker):
    specifier = rhythm_maker.default.duration_spellingSpecifier
else:
    specifier = rhythm_maker.duration_spellingSpecifier
if specifier is None:
    return True
if specifier.forbid_meter_rewriting:
    return False
return True

@staticmethod
def cleanup_logical_ties(music):
    for logical_tie in iterate(music).by_logical_tie(
        nontrivial=True, pitched=True, reverse=True):
        if len(logical_tie) != 2:
            continue
if not logical_tie.all_leaves_are_in_same_parent:
    continue
if logical_tie_written_duration == \durationtools.Duration(1, 8):
    mutate(logical_tie).replace([scoretools.Note("c'8")])
elif logical_tie_written_duration == \durationtools.Duration(1, 16):
    mutate(logical_tie).replace([scoretools.Note("c'16")])

@staticmethod
def collect_attack_points(score):
    import consort
    attack_point_map = collections.OrderedDict()
    iterator = iterate(score).by_timeline(component_class=scoretools.Note)
    for note in iterator:
        logical_tie = inspect(note).get_logical_tie()
        if note is not logical_tie.head:
            continue
        attack_point_signature = \consort.AttackPointSignature.from_logical_tie(logical_tie)
        attack_point_map[logical_tie] = attack_point_signature
    return attack_point_map

@staticmethod
def consolidate_demultiplexed_timespans(demultiplexed_maquette):
    for voice_name in demultiplexed_maquette:
        timespans = demultiplexed_maquette[voice_name]
        consolidated_timespans = SegmentMaker.consolidate_timespans(timespans)
        demultiplexed_maquette[voice_name] = consolidated_timespans

@staticmethod
def consolidate_rests(music):
    r"""Consolidates non-tupleted rests into separate containers in 'music'.
    ""

    >>> import consort
    >>>
    >>> music = scoretools.Container(r'''...
    ... { r4 c'8 }
    ... \times 2/3 { d'4 r8 }
    ... { r4 e'4 f'4 r4 }
    ... { r4 g8 r8 }
    ... { r4 }
    ... { r4 }
    ... { a'4 \times 2/3 { b'4 r8 } }
    ... { c''4 r8 }
    ... '''}
    >>> print(format(music))
    {

>>> music = consort.SegmentMaker.consolidate_rests(music)
>>> print(format(music))
{
  {r4
c'8
}
\times 2/3 {
d'4
r8
}
{r4
e'4
f'4
r4
}
{r4
g8
r8
}
{r4

::

   >>> music = consort.SegmentMaker.consolidate_rests(music)
   >>> print(format(music))
   {
     {r4
c'8
}
\times 2/3 {
d'4
r8
}
{c''4
r8
}


Returns 'music'.

```
prototype = {
    scoretools.Rest,
    scoretools.MultimeasureRest,
}
initial_music_duration = inspect_(music).get_duration()
initial_leaves = music.select_leaves()
if not isinstance(music[0], scoretools.Tuplet):
    leading_silence = scoretools.Container()
    while music[0] and isinstance(music[0][0], prototype):
        leading_silence.append(music[0].pop(0))
    if leading_silence:
        music.insert(0, leading_silence)
if not isinstance(music[-1], scoretools.Tuplet):
    tailing_silence = scoretools.Container()
    while music[-1] and isinstance(music[-1][-1], prototype):
        tailing_silence.insert(0, music[-1].pop())
    if tailing_silence:
        music.append(tailing_silence)
if len(music) < 2:
```
```python
return music

indices = reversed(range(len(music) - 1))

for index in indices:
    division = music[index]
    next_division = music[index + 1]
    silence = scoretools.Container()
    if not isinstance(division, scoretools.Tuplet):
        while division and isinstance(division[-1], prototype):
            silence.insert(0, division.pop())
    if not isinstance(next_division, scoretools.Tuplet):
        while next_division and \
            isinstance(next_division[0], prototype):
            silence.append(next_division.pop(0))
    if silence:
        music.insert(index + 1, silence)
    if not division:
        music.remove(division)
    if not next_division:
        music.remove(next_division)

for division in music[:]:
    if not division:
        music.remove(division)

assert inspect_(music).get_duration() == initial_music_duration
assert music.select_leaves() == initial_leaves

return music

@staticmethod
def consolidate_timespans(timespans, allow_silences=False):
    r'''Consolidates contiguous performed timespans by music specifier.

    >>> import consort

    >>> timespans = timespantools.TimespanInventory(
    ...     consort.PerformedTimespan(
    ...         start_offset=0,
    ...         stop_offset=10,
    ...         music_specifier='foo',
    ...     ),
    ...     consort.PerformedTimespan(
    ...         start_offset=10,
    ...         stop_offset=20,
    ...         music_specifier='foo',
    ...     ),
    ...     consort.PerformedTimespan(
    ...         start_offset=20,
    ...         stop_offset=25,
    ...         music_specifier='bar',
    ...     ),
    ...     consort.PerformedTimespan(
    ...         start_offset=40,
    ...     )
    ... )
    >>> import consort
    >>> timespans = timespantools.TimespanInventory([
    ...     consort.PerformedTimespan(
    ...         start_offset=0,
    ...         stop_offset=10,
    ...         music_specifier='foo',
    ...     ),
    ...     consort.PerformedTimespan(
    ...         start_offset=10,
    ...         stop_offset=20,
    ...         music_specifier='foo',
    ...     ),
    ...     consort.PerformedTimespan(
    ...         start_offset=20,
    ...         stop_offset=25,
    ...         music_specifier='bar',
    ...     ),
    ...     consort.PerformedTimespan(
    ...         start_offset=40,
    ...     )
    ... )
    >>> timespans = timespantools.TimespanInventory([
    ...     consort.PerformedTimespan(
    ...         start_offset=0,
    ...         stop_offset=10,
    ...         music_specifier='foo',
    ...     ),
    ...     consort.PerformedTimespan(
    ...         start_offset=10,
    ...         stop_offset=20,
    ...         music_specifier='foo',
    ...     ),
    ...     consort.PerformedTimespan(
    ...         start_offset=20,
    ...         stop_offset=25,
    ...         music_specifier='bar',
    ...     ),
    ...     consort.PerformedTimespan(
    ...         start_offset=40,
    ...     )
    ... )
```
... stop_offset=50,
... music_specifier='bar'),
... ),
... consort.PerformedTimespan(
... start_offset=50,
... stop_offset=58,
... music_specifier='bar'),
... ),
... ])
>>> print(format(timespans))
timespantools.TimespanInventory(

    [ consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(10, 1),
        music_specifier='foo',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(10, 1),
        stop_offset=durationtools.Offset(20, 1),
        music_specifier='foo',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(20, 1),
        stop_offset=durationtools.Offset(25, 1),
        music_specifier='bar',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(40, 1),
        stop_offset=durationtools.Offset(50, 1),
        music_specifier='bar',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(50, 1),
        stop_offset=durationtools.Offset(58, 1),
        music_specifier='bar',
    ),
    ]
)

::

>>> timespans = consort.SegmentMaker.consolidate_timespans(
    ... timespans)
>>> print(format(timespans))
timespantools.TimespanInventory(

    [ consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(20, 1),
        divisions=(
            durationtools.Duration(10, 1),
            durationtools.Duration(10, 1),
            ),
music_specifier='foo',
)
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(20, 1),
    stop_offset=durationtools.Offset(25, 1),
    divisions=(
        durationtools.Duration(5, 1),
        ),
    music_specifier='bar',
),
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(40, 1),
    stop_offset=durationtools.Offset(58, 1),
    divisions=(
        durationtools.Duration(10, 1),
        durationtools.Duration(8, 1),
        ),
    music_specifier='bar',
),
]
)

Returns new timespan inventory.

consolidated_timespans = timespantools.TimespanInventory()
for music_specifier, grouped_timespans in \ 
    SegmentMaker.group_timespans(timespans):
    if music_specifier is None and not allow_silences:
        continue
    if hasattr(music_specifier, 'minimum_phrase_duration'):
        duration = music_specifier.minimum_phrase_duration
        if duration and grouped_timespans.duration < duration:
            continue
        divisions = tuple(_duration for _ in grouped_timespans)
    first_timespan = grouped_timespans[0]
    last_timespan = grouped_timespans[-1]
    consolidated_timespan = new(
        first_timespan,
        divisions=divisions,
        stop_offset=last_timespan.stop_offset,
        original_stop_offset=last_timespan.original_stop_offset,
    )
    consolidated_timespans.append(consolidated_timespan)
consolidated_timespans.sort()
return consolidated_timespans

@staticmethod
def debug_timespans(timespans):
    import consort
    if not timespans:
        Consort.debug('No timespans found.')
    else:
        Consort.debug('DEBUG: Dumping timespans: ')
    if isinstance(timespans, dict):
for voice_name in timespans:
    consort.debug(' \t ' + voice_name)
    for timespan in timespans[voice_name]:
        divisions = timespan.divisions or []
        divisions = ' '.join(str(_) for _ in divisions)
        consort.debug(' \t \t {}: [{} ... {}] [{}, {}] [{}]').format(
            type(timespan).__name__,
            timespan.start_offset,
            timespan.stop_offset,
            timespan.duration,
            divisions,
            timespan.music,
        )

else:
    for timespan in timespans:
        consort.debug(' \t {}: [{}, to {}]').format(
            timespan.voice_name,
            type(timespan).__name__,
            timespan.start_offset,
            timespan.stop_offset,
        )

@staticmethod
def resolve_maquette(multiplexed_timespans):
    import consort
    demultiplexed_maquette = consort.TimespanInventoryMapping()
    for timespan in multiplexed_timespans:
        voice_name, layer = timespan.voice_name, timespan.layer
        if voice_name not in demultiplexed_maquette:
            demultiplexed_maquette[voice_name] = {}
        if layer not in demultiplexed_maquette[voice_name]:
            demultiplexed_maquette[voice_name][layer] = \
            timespantools.TimespanInventory()
            demultiplexed_maquette[voice_name][layer].append(
                timespan)
        for voice_name in demultiplexed_maquette:
            for layer, timespans in \
                demultiplexed_maquette[voice_name].items():
                cleaned_layer = SegmentMaker.cleanup_maquette_layer(timespans)
                demultiplexed_maquette[voice_name][layer] = cleaned_layer

        for voice_name in demultiplexed_maquette:
            for timespan_inventory in demultiplexed_maquette[voice_name]:
                timespan_inventory = \
                SegmentMaker.resolve_timespan_inventories(
                    timespan_inventory)
                demultiplexed_maquette[voice_name] = timespan_inventory
    return demultiplexed_maquette

@staticmethod
def cleanup_maquette_layer(timespans):
    import consort
    performed_timespans = timespantools.TimespanInventory()
    silent_timespans = timespantools.TimespanInventory()
    for timespan in timespans:
if isinstance(timespan, consort.PerformedTimespan):
    performed_timespans.append(timespan)
elif isinstance(timespan, consort.SilentTimespan):
    silent_timespans.append(timespan)
else:
    raise ValueError(timespan)
silent_timespans.compute_logical_or()
for performed_timespan in performed_timespans:
    silent_timespans - performed_timespan
    performed_timespans.extend(silent_timespans)
    performed_timespans.sort()
return performed_timespans

@staticmethod
def division_is_silent(division):
    r'''Is true when division only contains rests, at any depth.
    ::
    >>> import consort
    ::
    >>> division = scoretools.Container("c'4 d'4 e'4 f'4")
    >>> consort.SegmentMaker.division_is_silent(division)
    False
    ::
    >>> division = scoretools.Container('r4 r8 r16 r32')
    >>> consort.SegmentMaker.division_is_silent(division)
    True
    ::
    >>> division = scoretools.Container(
    ... r"c'4 \times 2/3 { d'8 r8 e'8 } f'4")
    >>> consort.SegmentMaker.division_is_silent(division)
    False
    ::
    >>> division = scoretools.Container(
    ... r'\times 2/3 { r4 \times 2/3 { r8. } }')
    >>> consort.SegmentMaker.division_is_silent(division)
    True
    Returns boolean.
    '"
    rest_prototype = {
        scoretools.Rest,
        scoretools.MultimeasureRest,
    }
    leaves = division.select_leaves()
return all(isinstance(leaf, rest_prototype) for leaf in leaves)

def interpret_rhythms(
    self,
    verbose=True,
):
    multiplexed_timespans = timespantools.TimespanInventory()

    with systemtools.Timer(
        enter_message='populating independent timespans:',
        exit_message='total:',
        verbose=verbose,
    ):
        meters, measure_offsets, multiplexed_timespans = \
        self.populate_independent_timespans(
            self.discard_final_silence,
            multiplexed_timespans,
            self.permitted_time_signatures,
            self.score,
            self.score_template,
            self.settings or (),
            self.desired_duration,
            self.timespan_quantization,
            verbose=verbose,
        )
        self._meters = meters

    with systemtools.Timer(
        enter_message='populating dependent timespans:',
        exit_message='total:',
        verbose=verbose,
    ):
        demultiplexed_maquette = \
        self.populate_dependent_timespans(
            self.measure_offsets,
            multiplexed_timespans,
            self.score,
            self.score_template,
            self.settings or (),
            self.desired_duration,
            verbose=verbose,
        )

    with systemtools.Timer(
        'populated silent timespans:,'
    ):
        demultiplexed_maquette = self.populate_silent_timespans(
            demultiplexed_maquette,
            self.measure_offsets,
            self.voice_names,
        )

    with systemtools.Timer(
        'populating silent timespans:',
        verbose=verbose,
    ):
' validated timespans:',
verbose=verbose,
):
    self.validate_timespans(demultiplexed_maquette)

with systemtools.Timer(
    enter_message=' rewriting meters:',
    exit_message=' total:',
    verbose=verbose,
):
    #expr = 'self rewrite_meters(demultiplexed_maquette, self.meters)'
    #systemtools.IOManager.profile_expr(
        # expr,
        # global_context=globals(),
        # local_context=locals(),
        # )
    self.rewrite_meters(
        demultiplexed_maquette,
        self.meters,
        self.score,
        verbose=verbose,
    )

with systemtools.Timer(
    ' populated score:',
    verbose=verbose,
):
    self.populate_score(
        demultiplexed_maquette,
        self.score,
    )

self._voicewise_timespans = demultiplexed_maquette

def find_meters(
    self,
    permitted_time_signatures=None,
    desired_duration=None,
    timespan_inventory=None,
):
    import consort
    offset_counter = metertools.OffsetCounter()
    for timespan in timespan_inventory:
        if isinstance(timespan, consort.SilentTimespan):
            continue
        offset_counter[timespan.start_offset] += 2
        offset_counter[timespan.stop_offset] += 1
        maximum = 1
        if offset_counter:
            maximum = int(max(offset_counter.values()))
        offset_counter[desired_duration] = maximum * 2
        maximum_meter_run_length = self.maximum_meter_run_length
        meters = metertools.Meter.fit_meters_to_expr(
            expr=offset_counter,
meters=permitted_time_signatures,
    maximum_run_length=maximum_meter_run_length,
)
    return tuple(meters)

@staticmethod
def get_rhythm_maker(music_specifier):
    import consort
    beam_specifier = rhythmmakertools.BeamSpecifier(
        beam_each_division=False,
        beam_divisions_together=False,
    )
    if music_specifier is None:
        rhythm_maker = rhythmmakertools.NoteRhythmMaker(
            beam_specifier=beam_specifier,
            output_masks=[rhythmmakertools.silence_all()],
        )
    elif music_specifier.rhythm_maker is None:
        rhythm_maker = rhythmmakertools.NoteRhythmMaker(
            beam_specifier=beam_specifier,
            tie_specifier=rhythmmakertools.TieSpecifier(
                tie_across_divisions=True,
            ),
        )
    elif isinstance(music_specifier.rhythm_maker,
                     consort.CompositeRhythmMaker):
        rhythm_maker = music_specifier.rhythm_maker.new(
            beam_specifier=beam_specifier,
        )
    else:
        rhythm_maker = music_specifier.rhythm_maker
        beam_specifier = rhythm_maker.beam_specifier or beam_specifier
        beam_specifier = new(
            beam_specifier,
            beam_each_division=False,
            beam_divisions_together=False,
        )
        rhythm_maker = new(
            rhythm_maker,
            beam_specifier=beam_specifier,
        )
    assert rhythm_maker is not None
    return rhythm_maker

@staticmethod
def group_nonsilent_divisions(music):
    r'''Groups non-silent divisions together.

    Yields groups in reverse order.
    ::

    >>> import consort

    639
>>> divisions = []
>>> divisions.append(scoretools.Container('r4'))
>>> divisions.append(scoretools.Container("c'4"))
>>> divisions.append(scoretools.Container('r4 r4'))
>>> divisions.append(scoretools.Container("d'4 d'4"))
>>> divisions.append(scoretools.Container("e'4 e'4 e'4"))
>>> divisions.append(scoretools.Container('r4 r4 r4'))
>>> divisions.append(scoretools.Container("f'4 f'4 f'4 f'4"))

>>> for group in consort.SegmentMaker.group_nonsilent_divisions(
...   divisions):
...   print(group)
(Container("f'4 f'4 f'4 f'4"),)
(Container("d'4 d'4"), Container("e'4 e'4 e'4"))
(Container("c'4"),)

Returns generator.

@staticmethod
def group_timespans(timespans):
def grouper(timespan):
music_specifier = None
if isinstance(timespan, consort.PerformedTimespan):
music_specifier = timespan.music_specifier
if music_specifier is None:
music_specifier = consort.MusicSpecifier()
forbid_fusing = timespan.forbid_fusing
return music_specifier, forbid_fusing
import consort
for partitioned_timespans in timespans.partition(
   include_tangent_timespans=True):
   for key, grouped_timespans in itertools.groupby(
      partitioned_timespans, grouper):
      music_specifier, forbid_fusing = key
      if forbid_fusing:
         for timespan in grouped_timespans:
            group = timespantools.TimespanInventory([timespan])
            yield music_specifier, group
      else:
group = timespantools.TimespanInventory(
grouped_timespans)
yield musicSpecifier, group

@staticmethod
def inscribe_demultiplexed_timespans(
demultiplexed_maquette,
	score,
)

counter = collections.Counter()
voice_names = demultiplexed_maquette.keys()
voice_names = SegmentMaker.sort_voice_names(score, voice_names)

for voice_name in voice_names:
    inscribed_timespans = timespantools.TimespanInventory()
    uninscribed_timespans = demultiplexed_maquette[voice_name]

    for timespan in uninscribed_timespans:
        if timespan.music is None:
            musicSpecifier = timespan.musicSpecifier
            if musicSpecifier not in counter:
                if musicSpecifier is None:
                    seed = 0
                else:
                    seed = musicSpecifier.seed or 0
                counter[musicSpecifier] = seed
                seed = counter[musicSpecifier]
            result = SegmentMaker.inscribe_timespan(
                timespan,
                seed=seed,
            )
            inscribed_timespans.extend(result)
        # Negative rotation mimics advancing through a series.
        counter[musicSpecifier] -= 1

        else:
            inscribed_timespans.append(timespan)
    demultiplexed_maquette[voice_name] = inscribed_timespans

@staticmethod
def inscribe_timespan(timespan, seed=None):
    r'''Inscribes 'timespan'.
    ::
    >>> import consort
    >>> musicSpecifier = consort.MusicSpecifier(
    ... rhythm_maker=rhythmmakertools.NoteRhythmMaker(
    ... output_masks=[
    ... rhythmmakertools.SilenceMask(
    ... indices=[0],
    ... period=3,
    ... ),
    ... ],
    ... ),
    ... )
    '')
```python
>>> timespan = consort.PerformedTimespan(
...   divisions=[durationtools.Duration(1, 4)] * 7,
...   start_offset=0,
...   stop_offset=(7, 4),
...   music_specifier=music_specifier,
... )
>>> print(format(timespan))
consort.tools.PerformedTimespan(
    start_offset=durationtools.Offset(0, 1),
    stop_offset=durationtools.Offset(7, 4),
    divisions=(
        durationtools.Duration(1, 4),
        durationtools.Duration(1, 4),
        durationtools.Duration(1, 4),
        durationtools.Duration(1, 4),
        durationtools.Duration(1, 4),
        durationtools.Duration(1, 4),
        durationtools.Duration(1, 4),
    ),
    music_specifier=consort.tools.MusicSpecifier(
        rhythm_maker=rhythmmakertools.NoteRhythmMaker(
            output_masks=rhythmmakertools.BooleanPatternInventory(
                
            ),
        ),
    ),
)
::

>>> result = consort.SegmentMaker.inscribe_timespan(timespan)
>>> print(format(result))
timespantools.TimespanInventory(
    [
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(1, 4),
            stop_offset=durationtools.Offset(3, 4),
            music=scoretools.Container(
                "( c'4 ) ( c'4 )"
            ),
            music_specifier=consort.tools.MusicSpecifier(
                rhythm_maker=rhythmmakertools.NoteRhythmMaker(
                    output_masks=rhythmmakertools.BooleanPatternInventory(
                        
                    ),
                ),
            ),
        ),
    ],
)```
Returns timespan inventory.

```python
inscribed_timespans = timespantools.TimespanInventory()
rhythm_maker = SegmentMaker.get_rhythm_maker(timespan.musicSpecifier)
durations = timespan.divisions[:]
music = SegmentMaker.make_music(
    rhythm_maker,
    durations,
    seed,
)
assert inspect_(music).get_duration() == timespan.duration
for container, duration in zip(music, durations):
    assert inspect_(container).get_duration() == duration
music = SegmentMaker.consolidate_rests(music)
assert inspect_(music).get_duration() == timespan.duration
for group in SegmentMaker.group_nonsilent_divisions(music):
    start_offset = inspect_(group[0]).get_timespan().start_offset
    stop_offset = inspect_(group[-1]).get_timespan().stop_offset
    start_offset += timespan.start_offset
    stop_offset += timespan.start_offset
    container = scoretools.Container()
    container.extend(group)
```
beam = spannertools.GeneralizedBeam(
    durations=[division._get_duration() for division in music],
    include_long_duration_notes=False,
    include_long_duration_rests=False,
    isolated_nib_direction=None,
    use_stemlets=False,
)
attach(beam, container, name='beam')
for division in container:
    durations = [division._get_duration()]
    beam = spannertools.GeneralizedBeam(
        durations=durations,
        include_long_duration_notes=False,
        include_long_duration_rests=False,
        isolated_nib_direction=None,
        use_stemlets=True,
    )
    attach(beam, division)
attach(timespan.musicSpecifier, container, scope=scoretools.Voice)
inscribed_timespan = new(
    timespan,
    divisions=None,
    music=container,
    start_offset=start_offset,
    stop_offset=stop_offset,
)
assert inspect_(container).get_duration() == \n    inscribed_timespan.duration
assert inspect_(container).get_timespan().start_offset == 0
assert inspect_(container[0]).get_timespan().start_offset == 0
inscribed_timespans.append(inscribed_timespan)
inscribed_timespans.sort()
return inscribed_timespans

@staticmethod
def leaf_is_tied(leaf):
    prototype = spannertools.Tie
    leaf_tie = None
    if inspect_(leaf).get_spanners(prototype):
        leaf_tie = inspect_(leaf).get_spanner(prototype)
    else:
        return False
    next_leaf = inspect_(leaf).get_leaf()
    if next_leaf is not None:
        if inspect_(next_leaf).get_spanners(prototype):
            next_leaf_tie = inspect_(next_leaf).get_spanner(prototype)
        if leaf_tie is next_leaf_tie:
            return True
    return False

@staticmethod
def make_music(rhythm_maker, durations, seed=0):
    music = rhythm_maker(durations, rotation=seed)
    for i, division in enumerate(music):
if (len(division) == 1 and
    isinstance(division[0], scoretools.Tuplet)):
    music[i] = division[0]
else:
    music[i] = scoretools.Container(division)

music = scoretools.Container(music)
prototype = rhythmmakertools.AccelerandoRhythmMaker
if not isinstance(rhythm_maker, prototype):
    for division in music[:]:
        if (isinstance(division, scoretools.Tuplet) and
            division.multiplier == 1):
            mutate(division).swap(scoretools.Container())

return music

@staticmethod
def meters_to_offsets(meters):
    r'''Converts 'meters' to offsets.

>>> import consort

>>> meters = [...
    ... metertools.Meter((3, 4)),
    ... metertools.Meter((2, 4)),
    ... metertools.Meter((6, 8)),
    ... metertools.Meter((5, 16)),
    ... ]

>>> offsets = consort.SegmentMaker.meters_to_offsets(meters)
>>> for x in offsets:
>>> ... x
>>> ...
    ... Offset(0, 1)
    ... Offset(3, 4)
    ... Offset(5, 4)
    ... Offset(2, 1)
    ... Offset(37, 16)

Returns tuple of offsets.
'''
durations = [_.duration for _ in meters]
offsets = mathtools.cumulative_sums(durations)
offsets = [durationtools.Offset(_) for _ in offsets]
return tuple(offsets)
@staticmethod
def meters_to_timespans(meters):
    r'''Convert 'meters' into a collection of annotated timespans.

    >>> import consort
    >>>

    >>> meters = [
    ...     metertools.Meter((3, 4)),
    ...     metertools.Meter((2, 4)),
    ...     metertools.Meter((6, 8)),
    ...     metertools.Meter((5, 16)),
    ...
    ]
    >>>

    >>> timespans = consort.SegmentMaker.meters_to_timespans(meters)
    >>> print(format(timespans))
    consort.tools.TimespanCollection(
        [
            timespantools.AnnotatedTimespan(
                start_offset=durationtools.Offset(0, 1),
                stop_offset=durationtools.Offset(3, 4),
                annotation=metertools.Meter('3/4 (1/4 1/4 1/4)'),
            ),
            timespantools.AnnotatedTimespan(
                start_offset=durationtools.Offset(3, 4),
                stop_offset=durationtools.Offset(5, 4),
                annotation=metertools.Meter('2/4 (1/4 1/4)'),
            ),
            timespantools.AnnotatedTimespan(
                start_offset=durationtools.Offset(5, 4),
                stop_offset=durationtools.Offset(2, 1),
            ),
            timespantools.AnnotatedTimespan(
                start_offset=durationtools.Offset(2, 1),
                stop_offset=durationtools.Offset(37, 16),
                annotation=metertools.Meter('5/16 ((3/16 (1/16 1/16 1/16)) (2/16 (1/16 1/16)))'),
            ),
        ]
    )
Returns timespan collections.

```
import consort
timespans = consort.TimespanCollection()
offsets = SegmentMaker.meters_to_offsets(meters)
for i, meter in enumerate(meters):
    start_offset = offsets[i]
    stop_offset = offsets[i + 1]
    timespan = timespantools.AnnotatedTimespan(
        annotation=meter,
        start_offset=start_offset,
        stop_offset=stop_offset,
    )
    timespans.insert(timespan)
return timespans

@staticmethod
def multiplex_timespans(demultiplexed_maquette):
    r'''Multiplexes 'demultiplexed_maquette' into a single timespan inventory.

    >>> import consort
    >>> demultiplexed = {}
    >>> demultiplexed['foo'] = timespantools.TimespanInventory([
    ...     timespantools.Timespan(0, 10),
    ...     timespantools.Timespan(15, 30),
    ... ])
    >>> demultiplexed['bar'] = timespantools.TimespanInventory([
    ...     timespantools.Timespan(5, 15),
    ...     timespantools.Timespan(20, 35),
    ... ])
    >>> demultiplexed['baz'] = timespantools.TimespanInventory([
    ...     timespantools.Timespan(5, 40),
    ... ])

    >>> multiplexed = consort.SegmentMaker.multiplex_timespans(
    ...     demultiplexed)
    >>> print(format(multiplexed))
    timespantools.TimespanInventory(
        [timespantools.Timespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(10, 1),
        ),
        timespantools.Timespan(
            start_offset=durationtools.Offset(5, 1),
            stop_offset=durationtools.Offset(15, 1),
        ),
    ])
```
timespantools.Timespan(
    start_offset=durationtools.Offset(5, 1),
    stop_offset=durationtools.Offset(40, 1),
).

timespantools.Timespan(
    start_offset=durationtools.Offset(15, 1),
    stop_offset=durationtools.Offset(30, 1),
).

timespantools.Timespan(
    start_offset=durationtools.Offset(20, 1),
    stop_offset=durationtools.Offset(35, 1),
).
}

Returns timespan inventory.

```python
multiplexed_timespans = timespantools.TimespanInventory()
for timespans in demultiplexed_maquette.values():
    multiplexed_timespans.extend(timespans)
    multiplexed_timespans.sort()
return multiplexed_timespans
```
...split timespans:',
verbose=verbose,
):
    self.split_demultiplexed_timespans(
meter_offsets,
demultiplexed_maquette,
)
with systemtools.Timer(
    split timespans:',
verbose=verbose,
):
    for voice_name, timespans in demultiplexed_maquette.items():
        self.prune_short_timespans(timespans)
with systemtools.Timer(
    pruned short timespans:',
verbose=verbose,
):
    for voice_name, timespans in demultiplexed_maquette.items():
        self.prune_malformed_timespans(timespans)
with systemtools.Timer(
    pruned malformed timespans:',
verbose=verbose,
):
    self.consolidate_demultiplexed_timespans(
demultiplexed_maquette,
)
with systemtools.Timer(
    consolidated timespans:',
verbose=verbose,
):
    self.inscribe_demultiplexed_timespans(
demultiplexed_maquette,
score,
)
return demultiplexed_maquette

def populate_independent_timespans(
    self,
discard_final_silence,
multiplexed_timespans,
permitted_time_signatures,
score,
score_template,
settings,
desired_duration,
timespan_quantization,
verbose=True,
):
    with systemtools.Timer(
        populated timespans:',
verbose=verbose,
):
        SegmentMaker.populate_multiplexed_maquette(
dependent=False,
score=score,
score_template=score_template,
settings=settings,
desired_duration=desired_duration,
timespan_inventory=multiplexed_timespans,
timespan_quantization=timespan_quantization,
)

with systemtools.Timer(
    ' found meters:',
    verbose=verbose,
):
    meters = self.find_meters(
        permitted_time_signatures=permitted_time_signatures,
        desired_duration=desired_duration,
        timespan_inventory=multiplexed_timespans,
    )
meter_offsets = SegmentMaker.meters_to_offsets(meters)

with systemtools.Timer(
    ' demultiplexed timespans:',
    verbose=verbose,
):
    demultiplexed_maquette = SegmentMaker.resolve_maquette(
        multiplexed_timespans)

with systemtools.Timer(
    ' split timespans:',
    verbose=verbose,
):
    SegmentMaker.split_demultiplexed_timespans(
        meter_offsets,
        demultiplexed_maquette,
    )

# TODO: Determine best place for malformed timespan pruning.

with systemtools.Timer(
    ' pruned short timespans:',
    verbose=verbose,
):
    SegmentMaker.prune_short_timespans(multiplexed_timespans)

with systemtools.Timer(
    ' pruned malformed timespans:',
    verbose=verbose,
):
    for voice_name, timespans in demultiplexed_maquette.items():
        SegmentMaker.prune_malformed_timespans(timespans)

with systemtools.Timer(
    ' consolidated timespans:',
    verbose=verbose,
):
    SegmentMaker.consolidate_demultiplexed_timespans(
        demultiplexed_maquette,
    )

with systemtools.Timer(
    ' inscribed timespans:',
    verbose=verbose,
):
SegmentMaker.inscribe_demultiplexed_timespans(
    demultiplexed_maquette,
    score,
)

with systemtools.Timer(
    ' inscribed timespans:',
    verbose=verbose,
):
    demultiplexed_timespans = SegmentMaker.multiplex_timespans(
        demultiplexed_maquette)

# TODO: Why prune after consolidation?
with systemtools.Timer(
    ' multiplexed timespans:',
    verbose=verbose,
):
    multiplexed_timespans = SegmentMaker.multiplex_timespans(
        demultiplexed_maquette)
    with systemtools.Timer(
        ' pruned meters:',
        verbose=verbose,
    ):
        meters = SegmentMaker.prune_meters(
            discard_final_silence=
            multiplexed_timespans.stop_offset,
        )
        meter_offsets = SegmentMaker.meters_to_offsets(meters)
    return meters, meter_offsets, multiplexed_timespans

@staticmethod
def populate_multiplexed_maquette(
    dependent=False,
    score=None,
    score_template=None,
    settings=None,
    desired_duration=None,
    timespan_inventory=None,
    timespan_quantization=None,
):
    segment_timespan = timespantools.Timespan(0, desired_duration)
    if timespan_quantization is None:
        timespan_quantization = durationtools.Duration(1, 16)
    if timespan_inventory is None:
        timespan_inventory = timespantools.TimespanInventory()
    independent_settings = [setting for setting in settings
        if not setting.timespan_maker.is_dependent
    ]
    dependent_settings = [setting for setting in settings
        if setting.timespan_maker.is_dependent
    ]
    if dependent:
        settings = dependent_settings
        start_index = len(independent_settings)
    else:
        settings = independent_settings
        start_index = 0
    if not settings:
        return False
    for layer, music_setting in enumerate(settings, start_index):
        music_setting(
layer=layer,
score=score,
score_template=score_template,
segment_timespan=segment_timespan,
timespan_inventory=timespan_inventory,
timespan_quantization=timespan_quantization,
)
SegmentMaker.debug_timespans(timespan_inventory)
return True

@staticmethod
def populate_score(
demultiplexed_maquette,
score,
):
    for voice_name, timespans in demultiplexed_maquette.items():
        voice = score[voice_name]
        for timespan in timespans:
            assert timespan.duration == inspect_(timespan.music).get_duration()
        voice.append(timespan.music)
    return score

@staticmethod
def populate_silent_timespans(
demultiplexed_maquette,
meter_offsets,
voice_names=None,
):
    import consort
    silent_musicSpecifier = consort.MusicSpecifier()
    rhythm_maker = SegmentMaker.get_rhythm_maker(None)
    if voice_names is None:
        voice_names = demultiplexed_maquette.keys()
    else:
        voice_names = set(voice_names)
        voice_names.update(demultiplexed_maquette.keys())
    for voice_name in voice_names:
        if voice_name not in demultiplexed_maquette:
            demultiplexed_maquette[voice_name] =
            timespanTools.TimespanInventory()
        timespans = demultiplexed_maquette[voice_name]
silences = timespanTools.TimespanInventory([
            consort.SilentTimespan(
                start_offset=0,
                stop_offset=meter_offsets[-1],
                voice_name=voice_name,
            )
        ])  
silences = SegmentMaker.subtract_timespan_inventories(silences, timespans)
silences = SegmentMaker.split_timespans(meter_offsets, silences)
    for group in silences.partition(include_tangent_timespans=True):
        start_offset = group.start_offset,
stop_offset = group.stop_offset,
durations = [_.duration for _ in group]
silence = SegmentMaker.make_music(
    rhythm_maker,
    durations,
)
attach(silent_music_specifier, silence, scope=scoretools.Voice)
silent_timespan = consort.PerformedTimespan(
    music=silence,
    start_offset=start_offset,
    stop_offset=stop_offset,
    voice_name=voice_name,
)
timespans.append(silent_timespan)
timespans.sort()
return demultiplexed_maquette

@staticmethod
def prune_meters(
    discard_final_silence,
    meters,
    stop_offset,
):
    discard_final_silence = bool(discard_final_silence)
    if discard_final_silence and stop_offset:
        meters = list(meters)
        total_meter_durations = sum(_.duration for _ in meters[:-1])
        while stop_offset <= total_meter_durations:
            meters.pop()
        total_meter_durations = sum(_.duration for _ in meters[:-1])
    return tuple(meters)

@staticmethod
def prune_short_timespans(timespans):
    for timespan in timespans[:]:
        if timespan.minimum_duration and \
            timespan.duration < timespan.minimum_duration and \
            timespan.music is None:
            timespans.remove(timespan)

@staticmethod
def prune_malformed_timespans(timespans):
    for timespan in timespans[:]:
        if not timespan.is_well_formed:
            assert timespan.music is None
            timespans.remove(timespan)

@staticmethod
def report(timespan_inventory):
    print('REPORTING')
    for timespan in timespan_inventory:
        print(
            '\t',
            '{}:'.format(timespan.voice_name),
            )
@staticmethod
def resolve_timespan_inventories(
timespan_inventories=None,
):
    import consort
    timespan_inventories = [x[1] for x in
        sorted(timespan_inventories.items(),
            key=lambda item: item[0],
        )
    ]
    for timespan_inventory in timespan_inventories:
        assert timespan_inventory.all_are_nonoverlapping
        resolved_inventory = consort.TimespanCollection()
        for timespan in timespan_inventory:
            if isinstance(timespan, consort.SilentTimespan):
                continue
            resolved_inventory.insert(timespan)
        for timespan_inventory in timespan_inventories[1:]:
            resolved_inventory = SegmentMaker.subtract_timespan_inventories(
                resolved_inventory,
                timespan_inventory,
            )
            for timespan in resolved_inventory[:]:
                if timespan.minimum_duration and \
                    timespan.duration < timespan.minimum_duration:
                    resolved_inventory.remove(timespan)
            for timespan in timespan_inventory:
                if isinstance(timespan, consort.SilentTimespan):
                    continue
                resolved_inventory.append(timespan)
            resolved_inventory.sort()
        resolved_inventory = timespantools.TimespanInventory(
            resolved_inventory[:],
        )
    return resolved_inventory

@staticmethod
def rewrite_container_meter(
    container,
    meter_timespans,
    forbid_staff_lines_spanner=None,
):
    assert meter_timespans
    assert meter_timespans[0].start_offset <= \
        inspect_(container).get_timespan().start_offset
    last_leaf = container.select_leaves()[1]
    is_tied = SegmentMaker.leaf_is_tied(last_leaf)
container_timespan = inspect_(container).get_timespan()

if isinstance(container, scoretools.Tuplet):
    contents_duration = container._contents_duration
    meter = metertools.Meter(contents_duration)
    boundary_depth = 1
    if meter.numerator in (3, 4):
        boundary_depth = None
        mutate(container[:]).rewrite_meter(
            meter,
            boundary_depth=boundary_depth,
            maximum_dot_count=2,
        )

elif len(meter_timespans) == 1:
    container_timespan = inspect_(container).get_timespan()
    container_start_offset = container_timespan.start_offset
    container_stop_offset = container_timespan.stop_offset
    meter_timespan = meter_timespans[0]
    relative_meter_start_offset = meter_timespan.start_offset
    assert relative_meter_start_offset <= container_start_offset
    absolute_meter_stop_offset = (relative_meter_start_offset +
                                   container_start_offset +
                                   meter_timespan.duration)
    assert container_stop_offset <= absolute_meter_stop_offset
    if meter_timespan.is_congruent_to_timespan(container_timespan) \
       and SegmentMaker.division_is_silent(container):
        multimeasure_rest = scoretools.MultimeasureRest(1)
        duration = inspect_(container).get_duration()
        multiplier = durationtools.Multiplier(duration)
        attach(multiplier, multimeasure_rest)
        container[:] = [multimeasure_rest]
        if not forbid_staff_lines_spanner:
            previous_leaf = multimeasure_rest._get_leaf(-1)
            if isinstance(previous_leaf, scoretools.MultimeasureRest):
                staff_lines_spanner = \n                inspect_(previous_leaf).get_spanner(
                    spannertools.StaffLinesSpanner)
                components = staff_lines_spanner.components
                components = components + [multimeasure_rest]
                detach(staff_lines_spanner)
            else:
                staff_lines_spanner = spannertools.StaffLinesSpanner([0])
                components = [multimeasure_rest]
                attach(
                    staff_lines_spanner,
                    components,
                    name='staff_lines_spanner',
                )
        else:
            meter = meter_timespan.annotation
            meter_offset = meter_timespan.start_offset
            initial_offset = container_start_offset - meter_offset
            boundary_depth = 1
if meter.numerator in (3, 4):
    boundary_depth = None
mute(container[:]).rewrite_meter(
    meter,
    boundary_depth=boundary_depth,
    initial_offset=initial_offset,
    maximum_dot_count=2,
)
else:
    # TODO: handle bar-line-crossing containers
    raise AssertionError('Bar-line-crossing containers not permitted.')
if is_tied:
    last_leaf = container.select_leaves()[-1]
    next_leaf = inspect(last_leaf).get_leaf()
    selection = selectiontools.ContiguousSelection((
        last_leaf, next_leaf))
    selection._attach_tie_spanner_to_leaf_pair()

@staticmethod
def rewrite_meters(
    demultiplexed_maquette,
    meters,
    score,  
    verbose=True,
):
    import consort
    meter_timespans = SegmentMaker.meters_to_timespans(meters)
    cache = {}
    for context_name in sorted(demultiplexed_maquette):
        inscribed_timespans = demultiplexed_maquette[context_name]
        consort.debug('CONTEXT: {}'.format(context_name))
        context = score[context_name]
        forbid_staff_lines_spanner = context.context_name == 'Dynamics'
        progress_indicator = systemtools.ProgressIndicator(
            message=' rewriting {}' .format(context_name),
            verbose=verbose,
        )
        with progress_indicator:
            for inscribed_timespan in inscribed_timespans:
                consort.debug('	|s|s|s'.format(
                    inscribed_timespan.start_offset,
                    inscribed_timespan.stop_offset,
                    inscribed_timespan.music,
                ))
                if not SegmentMaker.can_rewrite_meter(inscribed_timespan):
                    continue
                with systemtools.ForbidUpdate(
                    inscribed_timespan.music,
                    update_on_exit=True,
                ):  
                    for i, container in enumerate(inscribed_timespan.music):
                        container_timespan = inspect_(container).get_timespan()
                        container_timespan = container_timespan.translate(
                            inscribed_timespan.start_offset)
if i == 0:
    assert container_timespan.start_offset == \n        inscribed_timespan.start_offset
if i == (len(inscribed_timespan.music) - 1):
    assert container_timespan.stop_offset == \n        inscribed_timespan.stop_offset
if container_timespan in cache:
    intersecting_meters = cache[container_timespan]
else:
    intersecting_meters = \n        meter_timespans.find_timespans_intersecting_timespan(\n            container_timespan)
    cache[container_timespan] = intersecting_meters
    shifted_intersecting_meters = [\n        _.translate(-1 * inscribed_timespan.start_offset)\n        for _ in intersecting_meters\n    ]
    consort.debug(f'\t	{!r} {!r}'.format(\n        container,
        container_timespan,
    )
    for intersecting_meter in intersecting_meters:
        consort.debug(f'\t		{repr(intersecting_meter)}')
    SegmentMaker.rewrite_container_meter(\n        container,
        shifted_intersecting_meters,
        forbid_staff_lines_spanner,
    )
    SegmentMaker.cleanup_logical_ties(container)
    progress_indicator.advance()

@staticmethod
def sort_voice_names(score, voice_names):
    result = []
    for voice in iterate(score).by_class(scoretools.Voice):
        if voice.name in voice_names:
            result.append(voice.name)
    return tuple(result)

@staticmethod
def split_demultiplexed_timespans(\n    meter_offsets=None,\n    demultiplexed_maquette=None,\n):
    for voice_name in demultiplexed_maquette:
        timespan_inventory = demultiplexed_maquette[voice_name]
        split_inventory = SegmentMaker.split_timespans(\n            meter_offsets,\n            timespan_inventory,\n        )
        demultiplexed_maquette[voice_name] = split_inventory

@staticmethod
def split_timespans(offsets, timespan_inventory):
offsets = list(offsets)
timespan_inventory.sort()
split_inventory = timespantools.TimespanInventory()

for timespan in timespan_inventory:
    current_offsets = []
    while offsets and offsets[0] <= timespan.start_offset:
        offsets.pop(0)
    while offsets and offsets[0] < timespan.stop_offset:
        current_offsets.append(offsets.pop(0))
    if hasattr(timespan, 'music') and timespan.music:
        # We don't need to split already-inscribed timespans
        split_inventory.append(timespan)
        continue
    elif timespan.forbid_splitting:
        continue
    if current_offsets:
        shards = timespan.split_at_offsets(current_offsets)
        for shard in shards:
            if shard.minimum_duration:
                if shard.minimum_duration <= shard.duration:
                    split_inventory.append(shard)
            else:
                split_inventory.append(shard)
        else:
            if timespan.minimum_duration:
                if timespan.minimum_duration <= timespan.duration:
                    split_inventory.append(timespan)
            else:
                split_inventory.append(timespan)
    split_inventory.sort()
return split_inventory

@staticmethod
def subtract_timespan_inventories(inventory_one, inventory_two):
    r'''Subtracts ‘inventory_two’ from ‘inventory_one’.
    ::

    >>> inventory_one = timespantools.TimespanInventory([
    ...     timespantools.Timespan(0, 10),
    ...     timespantools.Timespan(10, 20),
    ...     timespantools.Timespan(40, 80),
    ...     timespantools.Timespan(80, 100),
    ...    ])

    ::

    >>> inventory_two = timespantools.TimespanInventory([
    ...     timespantools.Timespan(5, 15),
    ...     timespantools.Timespan(25, 35),
    ...     timespantools.Timespan(35, 45),
    ...     timespantools.Timespan(55, 65),
    ...     timespantools.Timespan(85, 95),
    ...    ])"""
>>> import consort
>>> manager = consort.SegmentMaker
>>> result = manager.subtract_timespan_inventories(
...     inventory_one,
...     inventory_two,
... )
>>> 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(15, 1),
        stop_offset=durationtools.Offset(20, 1),
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(45, 1),
        stop_offset=durationtools.Offset(55, 1),
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(65, 1),
        stop_offset=durationtools.Offset(80, 1),
    ),
])

>>> result = manager.subtract_timespan_inventories(
...     inventory_two,
...     inventory_one,
... )
>>> print(format(result))
timespantools.TimespanInventory([
    timespantools.Timespan(
        start_offset=durationtools.Offset(25, 1),
        stop_offset=durationtools.Offset(35, 1),
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(35, 1),
        stop_offset=durationtools.Offset(40, 1),
    ),
    timespantools.Timespan(
        start_offset=durationtools.Offset(85, 1),
        stop_offset=durationtools.Offset(95, 1),
    ),
])
import consort
resulting_timespans = consort.TimespanCollection()
if not inventory_two:
    return timespantools.TimespanInventory(inventory_one)
elif not inventory_one:
    return timespantools.TimespanInventory()
subtractee_index = 0
subtractor_index = 0
subtractee = None
subtractor = None
subtractee_is_modified = False
while subtractee_index < len(inventory_one) and \
    subtractor_index < len(inventory_two):
    if subtractee is None:
        subtractee = inventory_one[subtractee_index]
        subtractee_is_modified = False
    if subtractor is None:
        subtractor = inventory_two[subtractor_index]
    if subtractee.intersects_timespan(subtractor):
        subtraction = subtractee - subtractor
        if len(subtraction) == 1:
            subtractee = subtraction[0]
            subtractee_is_modified = True
        elif len(subtraction) == 2:
            resulting_timespans.insert(subtraction[0])
            subtractee = subtraction[1]
            subtractee_is_modified = True
        else:
            subtractee = None
            subtractee_index += 1
    else:
        if subtractee.stops_before_or_at_offset(\
            subtractor.start_offset):
            resulting_timespans.insert(subtractee)
            subtractee = None
            subtractee_index += 1
        else:
            subtractor = None
            subtractor_index += 1
    if subtractee_is_modified:
        if subtractee:
            resulting_timespans.insert(subtractee)
            resulting_timespans.insert(inventory_one[subtractee_index + 1:])
    else:
        resulting_timespans.insert(inventory_one[subtractee_index:])
resulting_timespans = timespantools.TimespanInventory(\
    resulting_timespans[:])
return resulting_timespans

@staticmethod
def validate_timespans(demultiplexed_maquette):
durations = set()
for voice_name, timespans in demultiplexed_maquette.items():

timespans.sort()
assert timespans.start_offset == 0
assert timespans.all_are_contiguous
assert timespans.all_are_well_formed
assert timespans.all_are_nonoverlapping
durations.add(timespans.stop_offset)
assert len(tuple(durations)) == 1

def update_segment_metadata(self):
    self._segment_metadata.update(
        end_instruments_by_staff=self.get_end_instruments(),
        end_tempo=self.get_end_tempo_indication(),
        end_time_signature=self.get_end_time_signature(),
        is_repeated=self.repeat,
        measure_count=len(self.meters),
    )

### PUBLIC PROPERTIES ###
@property
def attack_point_map(self):
    return self._attack_point_map

@property
def meters(self):
    return self._meters

@property
def score(self):
    return self._score

@property
def voicewise_timespans(self):
    return self._voicewise_timespans

@property
def desired_duration(self):
    tempo = self.tempo
    if tempo is None:
        tempo = indicatortools.Tempo((1, 4), 60)
    tempo_desired_duration_in_seconds = durationtools.Duration(
        tempo.duration_to_milliseconds(tempo.duration),
        1000,
    )
    desired_duration = durationtools.Duration(
        self.desired_duration_in_seconds /
        tempo_desired_duration_in_seconds
    ).limit_denominator(8))
    desired_duration *= tempo.duration
    count = desired_duration // durationtools.Duration(1, 8)
    desired_duration = durationtools.Duration(count, 8)
    assert 0 < desired_duration
    return desired_duration
@property
def desired_duration_in_seconds(self):
    return self._desired_duration_in_seconds

@desired_duration_in_seconds.setter
def desired_duration_in_seconds(self, desired_duration_in_seconds):
    if desired_duration_in_seconds is not None:
        desired_duration_in_seconds = durationtools.Duration(desired_duration_in_seconds,
                                                            )
    self._desired_duration_in_seconds = desired_duration_in_seconds

@property
def discard_final_silence(self):
    return self._discard_final_silence

@discard_final_silence.setter
def discard_final_silence(self, discard_final_silence):
    if discard_final_silence is not None:
        discard_final_silence = bool(discard_final_silence)
    self._discard_final_silence = discard_final_silence

@property
def final_markup(self):
    return None

@property
def annotate_colors(self):
    return self._annotate_colors

@annotate_colors.setter
def annotate_colors(self, expr):
    if expr is not None:
        expr = bool(expr)
    self._annotate_colors = expr

@property
def annotate_phrasing(self):
    return self._annotate_phrasing

@annotate_phrasing.setter
def annotate_phrasing(self, expr):
    if expr is not None:
        expr = bool(expr)
    self._annotate_phrasing = expr

@property
def annotate_timespans(self):
    return self._annotate_timespans

@annotate_timespans.setter
def annotate_timespans(self, expr):
    if expr is not None:
        expr = bool(expr)
    self._annotate_timespans = expr

@property
    def lilypond_file(self):
        return self._lilypond_file

@property
    def maximum_meter_run_length(self):
        return self._maximum_meter_run_length

    @maximum_meter_run_length.setter
    def maximum_meter_run_length(self, expr):
        if expr is not None:
            expr = int(expr)
            assert 0 < expr
            self._maximum_meter_run_length = expr

@property
    def measure_offsets(self):
        measure_durations = [x.duration for x in self.time_signatures]
        measure_offsets = mathtools.cumulative_sums(measure_durations)
        return measure_offsets

@property
    def name(self):
        return self._name

    @name.setter
    def name(self, expr):
        if expr is not None:
            expr = str(expr)
            self._name = expr

@property
    def omit_stylesheets(self):
        return self._omit_stylesheets

    @omit_stylesheets.setter
    def omit_stylesheets(self, omit_stylesheets):
        if omit_stylesheets is not None:
            omit_stylesheets = bool(omit_stylesheets)
            self._omit_stylesheets = omit_stylesheets

@property
    def permitted_time_signatures(self):
        r'''Gets and sets segment maker's permitted time signatures.

        ::

            >>> import consort
            >>> segment_maker = consort.SegmentMaker()
            >>> time_signatures = [(3, 4), (2, 4), (5, 8)]
            >>> segment_maker.permitted_time_signatures = time_signatures
            >>> print(format(segment_maker))
consort.tools.SegmentMaker(
    permitted_time_signatures=indicatortools.TimeSignatureInventory(
        [indicatortools.TimeSignature((3, 4)),
        indicatortools.TimeSignature((2, 4)),
        indicatortools.TimeSignature((5, 8))],
    ),
)

'''

return self._permitted_time_signatures

@permitted_time_signatures.setter
def permitted_time_signatures(self, permitted_time_signatures):
    if permitted_time_signatures is not None:
        permitted_time_signatures = indicatortools.TimeSignatureInventory(
            items=permitted_time_signatures,
        )
    self._permitted_time_signatures = permitted_time_signatures

@property
def score_package_metadata(self):
    module_name = '{}.__metadata__'.format(self.score_package_name)
    try:
        module = importlib.import_module(module_name)
        metadata = getattr(module, 'metadata')
    except ImportError:
        metadata = {}
    return metadata

@property
def score_package_module(self):
    module = importlib.import_module(self.score_package_name)
    return module

@property
def score_package_name(self):
    return 'consort'

@property
def score_package_path(self):
    return self.score_package_module.__path__[0]

@property
def score_template(self):
    r'''Gets and sets segment maker's score template.

    ::

      >>> import consort
      >>> segment_maker = consort.SegmentMaker()
      >>> score_template = templatetools.StringOrchestraScoreTemplate(
            ...         violin_count=2,
... viola_count=1,
... cello_count=1,
... contrabass_count=0,
... )

>>> segment_maker.score_template = score_template
>>> print(format(segment_maker))

consort.tools.SegmentMaker(
    score_template=templatetools.StringOrchestraScoreTemplate(
        violin_count=2,
        viola_count=1,
        cello_count=1,
        contrabass_count=0,
        split_hands=True,
        use_percussion_clefs=False,
    ),
)

...'''

    return self._score_template

@score_template.setter

def score_template(self, score_template):
    self._score_template = score_template

@property

def segment_duration(self):
    return sum(x.duration for x in self.time_signatures)

@property

def settings(self):
    return tuple(self._settings)

@settings.setter

def settings(self, settings):
    import consort
    if settings is not None:
        if not isinstance(settings, collections.Sequence):
            settings = (settings,)
        assert all(isinstance(_, consort.MusicSetting) for _ in settings)
        settings = list(settings)
        self._settings = settings or []

@property

def tempo(self):
    r'''Gets and sets segment maker tempo.

    ::

        >>> import consort
        >>> segment_maker = consort.SegmentMaker()
        >>> tempo = indicatortools.Tempo((1, 4), 52)
        >>> segment_maker.tempo = tempo
        >>> print(format(segment_maker))
    
    consort.tools.SegmentMaker(}
tempo = indicatortools.Tempo(
    duration=durationtools.Duration(1, 4),
    units_per_minute=52,
)

...,

tempo = self._tempo
if tempo is not None:
    return tempo
elif self._previous_segment_metadata is not None:
    tempo = self._previous_segment_metadata.get('end_tempo')
    if tempo:
        tempo = indicatortools.Tempo(*tempo)
    return tempo

@tempo.setter
def tempo(self, tempo):
    if tempo is not None:
        if not isinstance(tempo, indicatortools.Tempo):
            tempo = indicatortools.Tempo(tempo)
        self._tempo = tempo

@property
def time_signatures(self):
    return tuple(
        meter.implied_time_signature
        for meter in self.meters
    )

@property
def timespan_quantization(self):
    r'''Gets and sets segment maker timespan quantization.
    ::
    >>> import consort
    >>> segment_maker = consort.SegmentMaker()
    >>> timespan_quantization = (1, 8)
    >>> segment_maker.timespan_quantization = timespan_quantization
    >>> print(format(segment_maker))
    segment_maker = consort.tools.SegmentMaker(
        timespan_quantization=durationtools.Duration(1, 8),
    )
    ...,
    return self._timespan_quantization

@timespan_quantization.setter
def timespan_quantization(self, timespan_quantization):
    if timespan_quantization is not None:
        timespan_quantization = \
        durationtools.Duration(timespan_quantization)
class SilentTimespan(timespantools.Timespan):
    
    r'''A silent timespan.
    '''

    ### CLASS VARIABLES ###

    __slots__ = (
        '_layer',
        '_voice_name',
    )

    ### INITIALIZER ###

    def __init__(
        self,
        start_offset=mathtools.NegativeInfinity(),
        stop_offset=mathtools.Infinity(),
        layer=None,
        voice_name=None,
    ):  
        timespantools.Timespan.__init__(
            self,
            start_offset=start_offset,
            stop_offset=stop_offset,
        )
        if layer is not None:
            layer = int(layer)
            self._layer = layer
            self._voice_name = voice_name
### PRIVATE METHODS ###

```python
def _as_postscript(self, postscript_x_offset, postscript_y_offset, postscript_scale):
    start = (float(self.start_offset) * postscript_scale)
    start -= postscript_x_offset
    stop = (float(self.stop_offset) * postscript_scale)
    stop -= postscript_x_offset
    ps = markuptools.Postscript()
    ps.moveto(start, postscript_y_offset)
    ps.setdash([0.5])
    ps.lineto(stop, postscript_y_offset)
    ps.stroke()
    ps.moveto(start, postscript_y_offset + 0.75)
    ps.setdash()
    ps.lineto(stop, postscript_y_offset - 0.75)
    ps.stroke()
    ps.moveto(stop, postscript_y_offset + 0.75)
    ps.lineto(stop, postscript_y_offset - 0.75)
    ps.stroke()
    if self.layer is not None:
        ps.moveto(start, postscript_y_offset)
        ps.moveto(0.25, 0.5)
        #ps = ps.scale(0.8, 0.8)
        ps.show(str(self.layer))
        #ps = ps.scale(1.25, 1.25)
    return ps
```

### PUBLIC PROPERTIES ###

```python
@property
def forbid_fusing(self):
    return False

@property
def forbid_splitting(self):
    return False

@property
def is_left_broken(self):
    return False

@property
def is_right_broken(self):
    return False

@property
def layer(self):
    return self._layer
```
```python
@property
def minimum_duration(self):
    return 0

@property
def voice_name(self):
    return self._voice_name

class SimpleDynamicExpression(abctools.AbjadValueObject):
    r'''A dynamic expression.

    .. container:: example

        >>> import consort

        >>> dynamic_expression = consort.SimpleDynamicExpression(
        ...     hairpin_start_token='sfp',
        ...     hairpin_stop_token='o',
        ...     )

        >>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
        >>> dynamic_expression(staff[2:-2])
        >>> print(format(staff))

        \new Staff {
            c'8
            d'8
            \override Hairpin #'circled-tip = ##t
            e'8 \> \sfp
            f'8
            g'8
            a'8 !
            \revert Hairpin #'circled-tip
            b'8
            c''8
        }
```
.. container:: example

::

    >>> dynamic_expression = consort.SimpleDynamicExpression('f', 'p',)
    >>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
    >>> dynamic_expression(staff[2:-2])
    >>> print(format(staff))

    \new Staff {
    c'8
    d'8 \> \f
    f'8
    g'8
    a'8 \p
    b'8
    c''8
    }

    ...

    ### CLASS VARIABLES ###

    __slots__ =
    ('_hairpin_start_token',
    '_hairpin_stop_token',
    '_minimum_duration',
    )

    ### INITIALIZER ###

    def __init__(self,
                 hairpin_start_token='p',
                 hairpin_stop_token=None,
                 minimum_duration=durationtools.Duration(1, 4),
                ):
        lilypond_parser = lilypondparsertools.LilyPondParser
        known_dynamics = list(lilypond_parser.list_known_dynamics())
        known_dynamics.append('o')
        assert hairpin_start_token in known_dynamics, \
            (known_dynamics, hairpin_start_token)
        if hairpin_stop_token is not None:
            assert hairpin_stop_token in known_dynamics
        assert hairpin_start_token != 'o' or hairpin_stop_token != 'o'
        if hairpin_start_token == 'o':
            assert not hairpin_stop_token is None
        self._hairpin_start_token = hairpin_start_token
        self._hairpin_stop_token = hairpin_stop_token
        if minimum_duration is not None:
            minimum_duration = durationtools.Duration(minimum_duration)
self._minimum_duration = minimum_duration

### SPECIAL METHODS ###

def __call__(self, music, name=None):
    if not isinstance(music, selectiontools.SliceSelection):
        music = selectiontools.SliceSelection(music)
    is_short_group = False
    if len(music) < 2:
        is_short_group = True
    elif self.minimum_duration is not None:
        if music.get_duration() < self.minimum_duration:
            is_short_group = True
    instrument = inspect_(music[0]).get_effective(
        instrumenttools.Instrument,
    )
    logical_ties = tuple(iterate(music).by_logical_tie(pitched=True))
    if len(logical_ties) < 3:
        if instrument == instrumenttools.Piano() or 
            instrument == instrumenttools.Percussion():
            is_short_group = True
    grace_notes = None
    previous_leaf = inspect_(music[0]).get_leaf(-1)
    if previous_leaf is not None:
        graces = inspect_(previous_leaf).get_grace_containers('after')
        if graces:
            assert len(graces) == 1
            grace_notes = list(graces[0].select_leaves())
            music = selectiontools.ContiguousSelect(
                tuple(grace_notes) + tuple(music),
            )
    start_token = self.hairpin_start_token
    stop_token = self.hairpin_stop_token
    if is_short_group or stop_token is None:
        if start_token == 'o':
            start_token = stop_token
        if start_token.startswith('fp'):
            start_token = start_token[1:]
        command = indicatortools.LilyPondCommand(start_token, 'right')
        attach(command, music[0], name=name)
        return
    start_ordinal = NegativeInfinity
    if start_token != 'o':
        start_ordinal = indicatortools.Dynamic.dynamic_name_to_dynamic_ordinal(
            start_token
        )
    stop_ordinal = NegativeInfinity
    if stop_token != 'o':
        stop_ordinal = indicatortools.Dynamic.dynamic_name_to_dynamic_ordinal(stop_token)
        items = []
    is_circled = False
    if start_ordinal < stop_ordinal:
        if start_token != 'o':
            items.append(start_token)
    else:
is_circled = True
items.append('<')
items.append(stop_token)
elif stop_ordinal < start_ordinal:
    items.append(start_token)
    items.append('>')
    if stop_token != 'o':
        items.append(stop_token)
    else:
        items.append('!!')
        is_circled = True
hairpin_descriptor = ' '.join(items)
hairpin = spannertools.Hairpin(
    descriptor=hairpin_descriptor,
    include_rests=False,
)
if is_circled:
    override(hairpin).hairpin.circled_tip = True
attach(hairpin, music, name=name)

### PUBLIC PROPERTIES ###

@property
def hairpin_start_token(self):
    return self._hairpin_start_token

@property
def hairpin_stop_token(self):
    return self._hairpin_stop_token

@property
def minimum_duration(self):
    return self._minimum_duration

A.49  consort.tools.StopTrillSpan

# -*- encoding: utf-8 -*-
from abjad import inspect_
from abjad.tools import abctools
from abjad.tools import scoretools
from abjad.tools import systemtools

class StopTrillSpan(abctools.AbjadValueObject):
    __slots__ = ()

def _get_lilypond_format_bundle(self, component):
    import consort
    parentage = inspect_(component).get_parentage()
    prototype = scoretools.GraceContainer
    grace_container = None
    for parent in parentage:
        if isinstance(parent, prototype):

672
A.50 **consort.tools.StringContactSpanner**

```python
# -*- encoding: utf-8 -*-
from __future__ import print_function
from abjad import inspect_
from abjad.tools import indicatortools
from abjad.tools import markuptools
from abjad.tools import spannertools
from abjad.tools import scoretools
from abjad.tools import lilypondnametools

class StringContactSpanner(spannertools.Spanner):
    r'''String contact spanner.
    
    ::
    
    >>> import consort
    >>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
    >>> attach(indicatortools.StringContactPoint('sul tasto'),
    ...    staff[2], scope=Staff)
    >>> attach(indicatortools.StringContactPoint('ordinario'),
    ...    staff[3], scope=Staff)
    >>> attach(indicatortools.StringContactPoint('pizzicato'),
    ...    staff[4], scope=Staff)
    >>> attach(indicatortools.StringContactPoint('ordinario'),
    ...    staff[5], scope=Staff)
    >>> attach(indicatortools.StringContactPoint('sul ponticello'),
    ...    staff[6], scope=Staff)
    >>> attach(consort.StringContactSpanner(), staff[:])
    .. doctest::
    
    >>> print(format(staff))
    \new Staff {
    c'8 ^ \markup {
    \vcenter
    \italic
    \caps
```
\once \override TextSpanner.arrow-width = 0.25
\once \override TextSpanner.bound-details.left-broken.text = ##f
\once \override TextSpanner.bound-details.left.stencil-align-dir-y = #center
\once \override TextSpanner.bound-details.left.text = \markup {
  \halign
  #0
  \halign
  #0
  \concat
  {
    \hspace
    #1.5
    \parenthesize
    \caps
    S.T.
    \hspace
    #1.5
  }
}
\once \override TextSpanner.bound-details.right-broken.padding = 0
\once \override TextSpanner.bound-details.right.arrow = ##t
\once \override TextSpanner.bound-details.right.padding = 0
\once \override TextSpanner.bound-details.right.stencil-align-dir-y = #center
\once \override TextSpanner.bound-details.right.text = \markup {
  \halign
  #0
  \halign
  #0
  \concat
  {
    \hspace
    #1.5
    \caps
    Ord.
    \hspace
    #1.5
  }
}
\once \override TextSpanner.dash-fraction = 0.25
\once \override TextSpanner.dash-period = 1
f'8 \startTextSpan
g'8 \stopTextSpan ^ \markup {
  \vcenter
  \italic
  \caps
  Ord.
}
a'8 ^ \markup {
  \vcenter
  \italic

\caps
  Pizz.
}
\once \override TextSpanner.arrow-width = 0.25
\once \override TextSpanner.bound-details.left-broken.text = ##f
\once \override TextSpanner.bound-details.left.stencil-align-dir-y = #center
\once \override TextSpanner.bound-details.left.text = \markup {
  \halign
    \halign
      \caps
        Ord.
        \hspace #1.5
        \caps
        \hspace #1.5
  }
}\once \override TextSpanner.bound-details.right-broken.padding = 0
\once \override TextSpanner.bound-details.right.arrow = ##t
\once \override TextSpanner.bound-details.right.padding = 0
\once \override TextSpanner.bound-details.right.stencil-align-dir-y = #center
\once \override TextSpanner.bound-details.right.text = \markup {
  \halign
    \halign
      \caps
        S.P.
        \hspace #1.5
  }
\once \override TextSpanner.dash-fraction = 0.25
\once \override TextSpanner.dash-period = 1
\stopTextSpan \startTextSpan
\concat
\hspace #1.5
\cap
\hspace #1.5
\cap
\hspace #1.5
b'8 \stopTextSpan \startTextSpan
\cap
\stopTextSpan
\cap

::

>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8")
>>> attach(indicatortools.StringContactPoint('ordinario'),
...     staff[0], scope=Staff)
>>> attach(indicatortools.StringContactPoint('sul tasto'),
...     staff[2], scope=Staff)
>>> attach(indicatortools.StringContactPoint('ordinario'),
...     staff
... staff[4], scope=Staff)
>>> attach(indicatortools.StringContactPoint('sul tasto'),
... staff[6], scope=Staff)
>>> attach(consort.StringContactSpanner(), staff[:])

.. doctest::

>>> print(format(staff))
\new Staff {
  \once \override TextSpanner.arrow-width = 0.25
  \once \override TextSpanner.bound-details.left-broken.text = ##f
  \once \override TextSpanner.bound-details.left.stencil-align-dir-y = #center
  \once \override TextSpanner.bound-details.left.text = \markup {
    \halign
    #0
    \halign
    #0
    \concat
    {
      \hspace
      #1.5
      \caps
      Ord.
      \hspace
      #1.5
    }
  }
  \once \override TextSpanner.bound-details.right-broken.padding = 0
  \once \override TextSpanner.bound-details.right.arrow = ##t
  \once \override TextSpanner.bound-details.right.padding = 5
  \once \override TextSpanner.bound-details.right.stencil-align-dir-y = #center
  \once \override TextSpanner.dash-fraction = 0.25
  \once \override TextSpanner.dash-period = 1
  c'8 \startTextSpan
d'8 \startTextSpan
  \once \override TextSpanner.arrow-width = 0.25
  \once \override TextSpanner.bound-details.left-broken.text = ##f
  \once \override TextSpanner.bound-details.left.stencil-align-dir-y = #center
  \once \override TextSpanner.bound-details.left.text = \markup {
    \halign
    #0
    \halign
    #0
    \concat
    {
      \hspace
      #1.5
      \caps
      S.T.
      \hspace
      #1.5
    }
  }
  \once \override TextSpanner.bound-details.right-broken.padding = 0
### CLASS VARIABLES ###

```
__slots__ = ()
```

### INITIALIZER ###

```
def __init__(
    self,
    overrides=None,
):
    spannertools.Spanner.__init__(
        self,
        overrides=overrides,
    )
```

### PRIVATE METHODS ###

```
def _get_annotations(self, leaf):
    import consort

    leaves = self._get_leaves()
    index = leaves.index(leaf)
    prototype = indicatortools.StringContactPoint
    agent = inspect_(leaf)
    pizzicato = indicatortools.StringContactPoint('pizzicato')

    next_attached = None
    for i in range(index + 1, len(leaves)):
        next_leaf = leaves[i]
        indicators = next_leaf._get_indicators(
            indicatortools.StringContactPoint,
        )
        if indicators:
            next_attached = indicators[0]
            break

    actually_attached = current_attached = None
    indicators = inspect_(leaf).get_indicators(prototype)
    if indicators:
        actually_attached = current_attached = indicators[0]
    if self._is_my_first_leaf(leaf) and current_attached is None:
        current_attached = next_attached

    next_different = None
    next_after_different = None
    next_next_different = None
    for i in range(index + 1, len(leaves)):
        next_leaf = leaves[i]
        indicators = next_leaf._get_indicators(
            indicatortools.StringContactPoint,
        )
        if indicators:
            indicator = indicators[0]
```
if next_different is not None:
    if next_after_different is None:
        next_after_different = indicator
    if indicator != next_different:
        next_next_different = indicator
        break
    if indicator != current_attached and next_different is None:
        next_different = indicator

n = -1
if actually_attached is None:
    n = 0
previous_effective = agent.get_effective(prototype, n=n)
previous_attached = None
for i in reversed(range(index)):
    previous_leaf = leaves[i]
    indicators = previous_leaf._get_indicators(
        indicatortools.StringContactPoint,
    )
    if indicators:
        previous_attached = indicators[0]
        break
    if current_attached is not None and 
        not self._is_my_first_leaf(leaf) and 
        previous_attached is None:
        previous_attached = current_attached

previous_different = None
for i in reversed(range(index)):
    previous_leaf = leaves[i]
    indicators = previous_leaf._get_indicators(
        indicatortools.StringContactPoint,
    )
    if indicators:
        indicator = indicators[0]
        if indicator != current_attached:
            previous_different = indicator

has_start_markup = False
if current_attached is not None and 
    next_attached is not None and 
    current_attached != pizzicato and 
    next_different != pizzicato and 
    current_attached != next_attached:
    has_start_markup = True

has_stop_markup = False
if current_attached is not None and 
    current_attached != pizzicato and 
    (next_next_different == pizzicato or next_next_different is None):
    has_stop_markup = True
elif current_attached is not None and 
    next_different == next_after_different:
    has_stop_markup = True
stops_text_spanner = False
if current_attached is not None and \
    previous_different is not None and \
    current_attached != pizzicato and \
    previous_different != pizzicato and \
    current_attached != previous_attached:
    stops_text_spanner = True

if self._is_my_first_leaf(leaf) and \
    current_attached is None and \
    previous_effective is not None:
    current_attached = previous_effective

is_cautionary = False
if current_attached and current_attached == previous_attached:
    is_cautionary = True
elif current_attached and current_attached == previous_effective:
    is_cautionary = True

current_markup = None
if current_attached is not None:
    current_markup = current_attached.markup
if current_attached == previousAttached == next_attached and \
    current_attached != pizzicato:
    current_markup = None
if current_attached == previous_effective and \
    next_attached is None and \
    current_attached != pizzicato and \
    not self._is_my_first_leaf(leaf):
    current_markup = None

if current_markup is not None:
    if is_cautionary:
        current_markup = current_markup.parenthesize()

results = (
    current_attached,
    current_markup,
    has_start_markup,
    has_stop_markup,
    is_cautionary,
    next_attached,
    next_different,
    previous_attached,
    previous_effective,
    stops_text_spanner,
)

consort.debug(leaf)
consort.debug('\t', 'actually_attached', actually_attached)
consort.debug('\t', 'current_attached', current_attached)
consort.debug('\t', 'current_markup', current_markup)
consort.debug('\t', 'has_start_markup', has_start_markup)
consort.debug('t', 'has_stop_markup', has_stop_markup)
consort.debug('t', 'is_cautionary', is_cautionary)
consort.debug('t', 'next_attached', next_attached)
consort.debug('t', 'next_different', next_different)
consort.debug('t', 'next_after_different', next_after_different)
consort.debug('t', 'next_next_different', next_next_different)
consort.debug('t', 'previous_attached', previous_attached)
consort.debug('t', 'previous_effective', previous_effective)
consort.debug('t', 'stops_text_spanner', stops_text_spanner)

return results

def _get_lilypond_format_bundle(self, leaf):
    import consort

    lilypond_format_bundle = self._get_basic_lilypond_format_bundle(leaf)
    if not isinstance(leaf, scoretools.Leaf):
        return lilypond_format_bundle

        current_attached,
        current_markup,
        has_start_markup,
        has_stop_markup,
        is_cautionary,
        next_attached,
        next_different,
        previous_attached,
        previous_effective,
        stops_text_spanner,
    ) = self._get_annotations(leaf)

    if current_markup is None:
        consort.debug('\tRETURNING+++++++++++++++')
        consort.debug()
        return lilypond_format_bundle

        if has_start_markup and has_stop_markup:
            self._add_segment_start_contributions(
                lilypond_format_bundle,
                start_markup=current_markup,
                stop_markup=next_different.markup,
            )
        elif has_start_markup:
            self._add_segment_start_contributions(
                lilypond_format_bundle,
                start_markup=current_markup,
            )
        if stops_text_spanner:
            self._add_segment_stop_contributions(lilypond_format_bundle)

        should_attach_markup = False
        if current_markup and \
            not has_start_markup and \

681
```python
not has_stop_markup:
    should_attach_markup = True
if current_markup and \
    previous_attached is None and \
    not has_start_markup:
    should_attach_markup = True
if current_markup and \
    current_attached == indicatortools.StringContactPoint('pizzicato '):
    should_attach_markup = True
if current_attached is not None and \
    current_attached == next_attached and \
    previous_attached != current_attached and \
    previous_attached is not None:
    should_attach_markup = False
consort.debug('Attaching???', should_attach_markup)
if should_attach_markup:
    current_markup = markuptools.Markup(current_markup, Up)
    current_markup = current_markup.italic()
    current_markup = current_markup.vcenter()
    lilypond_format_bundle.right.markup.append(current_markup)
    consort.debug(format(lilypond_format_bundle))
    consort.debug()
return lilypond_format_bundle

def _add_segment_start_contributions(
    self,
    lilypond_format_bundle,
    start_markup=None,
    stop_markup=None,
):
    right_padding = 5
    if stop_markup is not None:
        right_padding = 0
    line_segment = indicatortools.Arrow(
        dash_fraction=0.25,
        dash_period=1,
        right_padding=right_padding,
    )
    if start_markup is not None:
        start_markup = markuptools.Markup.concat([
            markuptools.Markup.hspace(1.5),
            start_markup,
            markuptools.Markup.hspace(1.5),
        ])
        start_markup = start_markup.halign(0)
    if stop_markup is not None:
        stop_markup = markuptools.Markup.concat([
            markuptools.Markup.hspace(1.5),
            stop_markup,
```
markuptools.Markup.hspace(1.5),
}
stop_markup = stop_markup.halign(0)

string = r'\startTextSpan'
lilypond_format_bundle.right.spanner_starts.append(string)
overrides = line_segment._get_lilypond_grob_overrides()
for override__ in overrides:
    override_string = r'\n'.join(override__._override_format_pieces)
lilypond_format_bundle.grob_overrides.append(override_string)
if start_markup:
    override_ = lilypondnametools.LilyPondGrobOverride(
        grob_name='TextSpanner',
        is_once=True,
        property_path={
            'bound-details',
            'left',
            'text',
        },
        value=start_markup.halign(0),
    )
    override_string = r'\n'.join(override__._override_format_pieces)
lilypond_format_bundle.grob_overrides.append(override_string)
if stop_markup:
    override_ = lilypondnametools.LilyPondGrobOverride(
        grob_name='TextSpanner',
        is_once=True,
        property_path={
            'bound-details',
            'right',
            'text',
        },
        value=stop_markup.halign(0),
    )
    override_string = r'\n'.join(override__._override_format_pieces)
lilypond_format_bundle.grob_overrides.append(override_string)

def _add_segment_stop_contributions(
    self,
    lilypond_format_bundle,
):
    string = r'\stopTextSpan'
lilypond_format_bundle.right.spanner_stops.append(string)

A.51 consort.tools.StringQuartetScoreTemplate

1 # -*- encoding: utf-8 -*-
2 from abjad import detach
3 from abjad import iterate
4 from abjad.tools import indicatortools
5 from abjad.tools import instrumenttools
6 from abjad.tools import markuptools
7 from abjad.tools import scoretools
8 from consort.tools.ScoreTemplate import ScoreTemplate
class StringQuartetScoreTemplate(ScoreTemplate):
    """A string quartet score template."
    ::

    >>> import consort
    >>> template = consort.StringQuartetScoreTemplate()
    >>> score = template()
    >>> print(format(score))

    \context Score = "String Quartet Score" <<
    \tag #'time
    \context TimeSignatureContext = "Time Signature Context" {
        \tag #'violin-1
        \context StringPerformerGroup = "Violin 1 Performer Group" \with {
            instrumentName = \markup {
                \hcenter-in
                #10
                "Violin 1"
            }
            shortInstrumentName = \markup {
                \hcenter-in
                #10
                "Vln. 1"
            }
        } <<
        \context BowingStaff = "Violin 1 Bowing Staff" {
            \clef "percussion"
            \context Voice = "Violin 1 Bowing Voice" {
            }
        }
        \context FingeringStaff = "Violin 1 Fingering Staff" {
            \clef "treble"
            \context Voice = "Violin 1 Fingering Voice" {
            }
        }
    }

    >>
    \tag #'violin-2
    \context StringPerformerGroup = "Violin 2 Performer Group" \with {
        instrumentName = \markup {
            \hcenter-in
            #10
            "Violin 2"
        }
        shortInstrumentName = \markup {
            \hcenter-in
            #10
            "Vln. 2"
        }
    } <<
    \context BowingStaff = "Violin 2 Bowing Staff" {
        \clef "percussion"
>>> for item in sorted(template.context_name_abbreviations.items()):
...     item
...     ('cello', 'Cello Performer Group')
...     ('cello_lh', 'Cello Fingering Voice')
...     ('cello_rh', 'Cello Bowing Voice')
...     ('viola', 'Viola Performer Group')
...     ('viola_lh', 'Viola Fingering Voice')
...     ('viola_rh', 'Viola Bowing Voice')
...     ('violin_1', 'Violin 1 Performer Group')
...     ('violin_1_lh', 'Violin 1 Fingering Voice')
...     ('violin_1_rh', 'Violin 1 Bowing Voice')
...     ('violin_2', 'Violin 2 Performer Group')
...     ('violin_2_lh', 'Violin 2 Fingering Voice')
...     ('violin_2_rh', 'Violin 2 Bowing Voice')

>>> for item in sorted(template.composite_context_pairs.items()):
...     item
...     ('cello', ('cello_rh', 'cello_lh'))
...     ('viola', ('viola_rh', 'viola_lh'))
...     ('violin_1', ('violin_1_rh', 'violin_1_lh'))
...     ('violin_2', ('violin_2_rh', 'violin_2_lh'))

>>> template = consort.StringQuartetScoreTemplate(split=False)
>>> score = template()
>>> print(format(score))
\context Score = "String Quartet Score" <<
\tag \#'time
\context TimeSignatureContext = "Time Signature Context" {
}
\tag \#'violin-1
\context StringPerformerGroup = "Violin 1 Performer Group" \with {
    instrumentName = \markup {
        \hspace \#10
        "Violin 1"
    }
    shortInstrumentName = \markup {
        \hspace \#10
        "Vln. 1"
    }
}
\context StringStaff = "Violin 1 Staff" {  
  \context Voice = "Violin 1 Voice" {  
    \clef "treble"  
  }  
}  

\tag #'violin-2
\context StringPerformerGroup = "Violin 2 Performer Group" \with {  
  instrumentName = \markup {  
    \hspace #10  
    "Violin 2"  
  }  
  shortInstrumentName = \markup {  
    \hspace #10  
    "Vln. 2"  
  }  
}  

\tag #'viola
\context StringPerformerGroup = "Viola Performer Group" \with {  
  instrumentName = \markup {  
    \hspace #10  
    Viola  
  }  
  shortInstrumentName = \markup {  
    \hspace #10  
    Va.  
  }  
}  

\tag #'cello
\context StringPerformerGroup = "Cello Performer Group" \with {  
  instrumentName = \markup {  
    \hspace #10  
    Cello  
  }  
  shortInstrumentName = \markup {  
    \hspace #10  
    Cello  
  }  
}
#10
Vc.
}
} <<
\context StringStaff = \"Cello Staff\" {
\context Voice = \"Cello Voice\" {
\clef \"bass\"
}
}
} >>
} >>
}::

>>> for item in sorted(template.context_name_abbreviations.items()):
... item
...
('cello', 'Cello Voice')
('viola', 'Viola Voice')
('violin_1', 'Violin 1 Voice')
('violin_2', 'Violin 2 Voice')
}
::

>>> for item in sorted(template.composite_context_pairs.items()):
... item
...
''

### CLASS VARIABLES ###

__slots__ = ('_split',
'_without_instruments',
)

### INITIALIZER ###

def __init__(self, split=True, without_instruments=None):
    if split is not None:
        split = bool(split)
    self._split = split
    self._without_instruments = without_instruments
    ScoreTemplate.__init__(self)

### SPECIAL METHODS ###

def __call__(self):
    import consort
    manager = consort.ScoreTemplateManager
time_signature_context = manager.make_time_signature_context()

violin_one = manager.make_single_string_performer(
    clef=indicatortools.Clef('treble'),
    instrument=instrumenttools.Violin(
        instrument_name='violin 1',
        instrument_name_markup=markuptools.Markup('Violin 1').hcenter_in(10),
        short_instrument_name='vln. 1',
        short_instrument_name_markup=markuptools.Markup('Vln. 1').hcenter_in(10)
    ),
    split=self.split,
    score_template=self,
)

violin_two = manager.make_single_string_performer(
    clef=indicatortools.Clef('treble'),
    instrument=instrumenttools.Violin(
        instrument_name='violin 2',
        instrument_name_markup=markuptools.Markup('Violin 2').hcenter_in(10),
        short_instrument_name='vln. 2',
        short_instrument_name_markup=markuptools.Markup('Vln. 2').hcenter_in(10)
    ),
    split=self.split,
    score_template=self,
)

viola = manager.make_single_string_performer(
    clef=indicatortools.Clef('alto'),
    instrument=instrumenttools.Viola(
        instrument_name='viola',
        instrument_name_markup=markuptools.Markup('Viola').hcenter_in(10),
        short_instrument_name='va.',
        short_instrument_name_markup=markuptools.Markup('Va.').hcenter_in(10)
    ),
    split=self.split,
    score_template=self,
)

cello = manager.make_single_string_performer(
    clef=indicatortools.Clef('bass'),
    instrument=instrumenttools.Cello(
        instrument_name='cello',
        instrument_name_markup=markuptools.Markup('Cello').hcenter_in(10),
        short_instrument_name='vc.',
        short_instrument_name_markup=markuptools.Markup('Vc.').hcenter_in(10)
    ),
)
A.52 \texttt{consort.tools.TaleaTimespanMaker}

```python
from abjad.tools import datastructuretools
from abjad.tools import durationtools
from abjad.tools import rhythmtools
from abjad.tools import timespantools
from consort.tools.TimespanMaker import TimespanMaker

class TaleaTimespanMaker(TimespanMaker):
    r'''A talea timespan maker.''

    >>> import consort
    >>> timespan_maker = consort.TaleaTimespanMaker(
    ...     initial_silence_talea=rhythmtools.Talea(
    ...         counts=(0, 4),
    ...         denominator=16,
    ...     )
    ... )
```
>>> print(format(timespan_maker))
consort.tools.TaleaTimespanMaker(
    initial_silence_talea=rhythmmakertools.Talea(
        counts=(0, 4),
        denominator=16,
    ),
    playing_talea=rhythmmakertools.Talea(
        counts=(4,),
        denominator=16,
    ),
    playing_groupings=(1,),
    repeat=True,
    silence_talea=rhythmmakertools.Talea(
        counts=(4,),
        denominator=16,
    ),
    step_anchor=Right,
    synchronize_groupings=False,
    synchronize_step=False,
)
::

>>> import collections
>>> music_specifiers = collections.OrderedDict([... ('Violin', None),
... ('Viola', None),
... ])
>>> target_timespan = timespantools.Timespan(0, 1)
>>> timespan_inventory = timespan_maker(
    ... music_specifiers=music_specifiers,
    ... target_timespan=target_timespan,
    ... )
>>> print(format(timespan_inventory))
timespantools.TimespanInventory([
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(1, 4),
        voice_name='Violin',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(1, 4),
        stop_offset=durationtools.Offset(1, 2),
        voice_name='Viola',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(1, 2),
        stop_offset=durationtools.Offset(3, 4),
        voice_name='Violin',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(3, 4),
    ),
]
stop_offset=durationtools.Offset(1, 1),
    voice_name='Viola',
),
]
)

::

>>> timespan_maker = new(timespan_maker,
    ... initial_silence_talea=None,
    ... synchronize_step=True,
    ... )

>>> timespan_inventory = timespan_maker(
    ... music_specifiers=music_specifiers,
    ... target_timespan=target_timespan,
    ... )

>>> print(format(timespan_inventory))
timespantools.TimespanInventory(
    [consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(1, 4),
        voice_name='Viola',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(0, 1),
        stop_offset=durationtools.Offset(1, 4),
        voice_name='Violin',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(1, 2),
        stop_offset=durationtools.Offset(3, 4),
        voice_name='Viola',
    ),
    consort.tools.PerformedTimespan(
        start_offset=durationtools.Offset(1, 2),
        stop_offset=durationtools.Offset(3, 4),
        voice_name='Violin',
    ),
    ]
)

::

>>> timespan_maker = new(timespan_maker,
    ... initial_silence_talea=rhythmmakertools.Talea(
        ... counts=(0, 2),
        ... denominator=16,
        ... ),
    ... )

>>> timespan_inventory = timespan_maker(
    ... music_specifiers=music_specifiers,
    ... target_timespan=target_timespan,
    ... )
>>> print(format(timespan_inventory))
timespantools.TimespanInventory(
    [
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(0, 1),
            stop_offset=durationtools.Offset(1, 4),
            voice_name='Violin',
        ),
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(1, 8),
            stop_offset=durationtools.Offset(3, 8),
            voice_name='Viola',
        ),
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(5, 8),
            stop_offset=durationtools.Offset(7, 8),
            voice_name='Violin',
        ),
        consort.tools.PerformedTimespan(
            start_offset=durationtools.Offset(3, 4),
            stop_offset=durationtools.Offset(1, 1),
            voice_name='Viola',
        ),
    ]
)
'''

### CLASS VARIABLES ###

__slots__ =
    (_fuse_groups',
    '_initial_silence_talea',
    '_padding',
    '_playing_talea',
    '_playing_groupings',
    '_reflect',
    '_repeat',
    '_silence_talea',
    '_step_anchor',
    '_synchronize_groupings',
    '_synchronize_step',
)

### INITIALIZER ###

def __init__(
    self,
    fuse_groups=None,
    initial_silence_talea=None,
    output_masks=None,
    padding=None,
    playing_talea=rhythmmakertools.Talea(counts=[4],
    ...
TimespanMaker.__init__(
    self,
    output_masks=output_masks,
    padding=padding,
    seed=seed,
    timespan_specifier=timespan_specifier,
)

if fuse_groups is not None:
    fuse_groups = bool(fuse_groups)
    self._fuse_groups = fuse_groups

if initial_silence_talea is not None:
    assert isinstance(initial_silence_talea, rhythmmakertools.Talea)
    assert all(0 <= x for x in initial_silence_talea.counts)
    self._initial_silence_talea = initial_silence_talea

assert isinstance(playing_talea, rhythmmakertools.Talea)
assert playing_talea.counts
assert all(0 < x for x in playing_talea.counts)
self._playing_talea = playing_talea

if not isinstance(playing_groupings, collections.Sequence):
    playing_groupings = (playing_groupings,)
playing_groupings = tuple(int(x) for x in playing_groupings)
assert len(playing_groupings)
assert all(0 < x for x in playing_groupings)
self._playing_groupings = playing_groupings

if reflect is not None:
    reflect = bool(reflect)
    self._reflect = reflect

self._repeat = bool(repeat)

if silence_talea is not None:
    assert isinstance(silence_talea, rhythmmakertools.Talea)
    assert silence_talea.counts

if step_anchor is not None:
    step_anchor = Right

if synchronize_groupings is not None:
    synchronize_groupings = False

if synchronize_step is not None:
    synchronize_step = False

if timespan_specifier is not None:
    timespan_specifier = None
assert all(0 <= x for x in silence_talea.counts)

self._silence_talea = silence_talea

assert step_anchor in (Left, Right)
self._step_anchor = step_anchor
self._synchronize_groupings = bool(synchronize_groupings)
self._synchronize_step = bool(synchronize_step)

### PRIVATE METHODS ###

def _make_infinite_iterator(self, sequence):
    index = 0
    sequence = datastructuretools.CyclicTuple(sequence)
    while True:
        yield sequence[index]
        index += 1

def _make_timespans(self, layer=None, music_specifiers=None, target_timespan=None, timespan_inventory=None):
    import consort
    initial_silence_talea = self.initial_silence_talea
    if not initial_silence_talea:
        initial_silence_talea = rhythmmakertools.Talea((0,), 1)
    initial_silence_talea = consort.Cursor(initial_silence_talea)
    playing_talea = consort.Cursor(self.playing_talea)
    playing_groupings = consort.Cursor(self.playing_groupings)
    silence_talea = self.silence_talea
    if silence_talea is None:
        silence_talea = rhythmmakertools.Talea((0,), 1)
    silence_talea = consort.Cursor(silence_talea)

    if self.seed is not None and 0 < self.seed:
        for _ in range(self.seed):
            next(initial_silence_talea)
            next(playing_talea)
            next(playing_groupings)
            next(silence_talea)

    if self.synchronize_step:
        procedure = self._make_with_synchronized_step
    else:
        procedure = self._make_without_synchronized_step
    new_timespan_inventory, final_offset = procedure(
        initial_silence_talea=initial_silence_talea,
        layer=layer,
        playing_talea=playing_talea,
        playing_groupings=playing_groupings,
        music_specifiers=music_specifiers,
        silence_talea=silence_talea,
target_timespan = target_timespan,
)
assert all(0 < _ duration for _ in new_timespan_inventory), \
(format(self), target_timespan)

if self.reflect:
    new_timespan_inventory = new_timespan_inventory.reflect(
        axis=target_timespan.axis,
    )

return new_timespan_inventory

def _make_with_synchronized_step(
    self,
    initial_silence_talea=None,
    layer=None,
    playing_talea=None,
    playing_groupings=None,
    music_specifiers=None,
    silence_talea=None,
    target_timespan=None,
):
    import consort

counter = collections.Counter()
timespan_inventory = timespantools.TimespanInventory()
start_offset = target_timespan.start_offset
stop_offset = target_timespan.stop_offset
can_continue = True

while start_offset < stop_offset and can_continue:
    silence_duration = next(silence_talea)
    durations = []
    if self.synchronize_groupings:
        grouping = next(playing_groupings)
        durations = [next(playing_talea) for _ in range(grouping)]
    for context_name, music_specifier in music_specifiers.items():
        if context_name not in counter:
            counter[context_name] = 0
            seed = counter[context_name]
            initial_silence_duration = next(initial_silence_talea)
            if not self.synchronize_groupings:
                grouping = next(playing_groupings)
            durations = [next(playing_talea) for _ in range(grouping)]
        maximum_offset = {
            start_offset +
            sum(durations) +
            silence_duration +
            initial_silence_duration
        }
        #if self.padding:
        #    maximum_offset += (self.padding * 2)
        maximum_offset = min(maximum_offset, stop_offset)
        if self.step_anchor is Left:
            maximum_offset = min(
                maximum_offset,
(  
    initial_silence_duration +  
    start_offset +  
    silence_duration  
  ),
)

  current_offset = start_offset + initial_silence_duration

  #if self.padding:
  #  current_offset += self.padding
  #  maximum_offset -= self.padding

  group_offset = current_offset

  valid_durations = []

  for duration in durations:
    if maximum_offset < (current_offset + duration):
      can_continue = False
      break
    valid_durations.append(duration)

  if self.fuse_groups:
    valid_durations = [sum(valid_durations)]

  new_timespans = musicSpecifier(
    durations=valid_durations,
    layer=layer,
    output_masks=self.output_masks,
    padding=self.padding,
    seed=seed,
    start_offset=group_offset,
    timespanSpecifier=self.timespanSpecifier,
    voice_name=context_name,
  )

  if all(isinstance(_, consort.SilentTimespan)
          for _ in new_timespans):
    for _ in new_timespans:
      new_timespans[:] = []
    timespan_inventory.extend(new_timespans)
    counter[context_name] += 1
    timespan_inventory.sort()
  if self.step_anchor == Right and timespan_inventory:
    start_offset = timespan_inventory.stop_offset
    start_offset += silence_duration
    if not self.repeat:
      break
    return timespan_inventory, start_offset

  def _make_without_synchronized_step(
    self,
    initial_silence_talea=None,  
    layer=None,  
    playing_talea=None,  
    playing_groupings=None,  
    music_specifiers=None,  
    silence_talea=None,  
    target_timespan=None,
import consort
counter = collections.Counter()
timespan_inventory = timespantools.TimespanInventory()
start_offset = target_timespan.start_offset
stop_offset = target_timespan.stop_offset
final_offset = durationtools.Offset(0)
for context_name, music_specifier in music_specifiers.items():
    if context_name not in counter:
        counter[context_name] = 0

    start_offset = target_timespan.start_offset
    start_offset += next(initial_silence_talea)
    can_continue = True

    while start_offset < stop_offset and can_continue:
        seed = counter[context_name]

        silence_duration = next(silence_talea)
        grouping = next(playing_groupings)
        durations = [next(playing_talea) for _ in range(grouping)]
        #if self.padding:
        #    start_offset += self.padding

        maximum_offset = start_offset + sum(durations) + 
        silence_duration
        maximum_offset = min(maximum_offset, stop_offset)
        if self.step_anchor is Left:
            maximum_offset = min(maximum_offset, 
            start_offset + silence_duration)
        #if self.padding:
        #    maximum_offset -= self.padding

        group_offset = current_offset = start_offset

        valid_durations = []
        for duration in durations:
            if maximum_offset < (current_offset + duration):
                can_continue = False
                break
        valid_durations.append(duration)
        current_offset += duration
        if len(durations) != len(valid_durations):
            for _ in range(len(durations) - len(valid_durations)):
                playing_talea.backtrack()
        if valid_durations and self.fuse_groups:
            valid_durations = [sum(valid_durations)]

        new_timespans = music_specifier( 
            durations=valid_durations,
            layer=layer,
            output_masks=self.output_masks,
padding=self.padding,
    seed=seed,
    start_offset=group_offset,
    timespanSpecifier=self.timespanSpecifier,
    voice_name=context_name,
}

if all(isinstance(_, consort.SilentTimespan)
    for _ in new_timespans):
    new_timespans = []
    timespan_inventory.extend(new_timespans)

if self.step_anchor is Left:
    start_offset += silence_duration
else:
    start_offset = current_offset + silence_duration

if stop_offset <= start_offset:
    can_continue = False

if not can_continue:
    if not valid_durations:
        silence_talea.backtrack()
        silence_talea.backtrack()
        playing_groupings.backtrack()

    if not self.repeat:
        break
    counter[context_name] += 1
if final_offset < start_offset:
    final_offset = start_offset
return timespan_inventory, final_offset

### PUBLIC PROPERTIES ###

@property
def fuse_groups(self):
    return self._fuse_groups

@property
def initial_silence_talea(self):
    return self._initial_silence_talea

@property
def playing_groupings(self):
    return self._playing_groupings

@property
def playing_talea(self):
    return self._playing_talea

@property
def reflect(self):
    return self._reflect
class TextSpannerExpression(abctools.AbjadValueObject):
    ### INITIALIZER ###
    def __init__(self, markup_tokens=None, transitions=None):
        if markup_tokens is not None:
            coerced_markup_tokens = []
            for x in markup_tokens:
                if isinstance(x, (markuptools.Markup, type(None))):
                    markup = x

A.53 consort.tools.TextSpannerExpression
elif hasattr(x, 'markup'):
    markup = x.markup
elif hasattr(x, '_get_markup'):
    markup = x._get_markup()
else:
    markup = markuptools.Markup(x)
coerced_markup_tokens.append(markup)
self._markup_tokens = markuptools.CyclicTuple(coerced_markup_tokens)

### SPECIAL METHODS ###

def __call__(self, music, name=None, seed=0):
    selections = self._get_selections(music)
    if 1 <= len(selections):
        for selection in selections[:-1]:
            markup, transition = self._get_attachments(seed)
            # do stuff
            seed += 1
    markup, transition = self._get_attachments(seed)
    selection = selections[-1]
    if selection.get_duration() <= durationtools.Duration(1, 8):
        # do stuff
        pass
    else:
        # do stuff
        seed += 1
        markup, transition = self._get_attachments(seed)
        # do stuff
        text_spanner = spannertools.TextSpanner()
        attach(text_spanner, music, name=name)

### PRIVATE METHODS ###

def _get_selections(self, music):
    selections = []
    for division in music:
        selection = division.select_leaves()
        selections.append(selection)
    return selections

### PUBLIC PROPERTIES ###

@property
def markup_tokens(self):
    return self._markup_tokens
@property
def transitions(self):
    return self._transitions

A.54  \texttt{consort.tools.TimespanCollection}

```python
# -*- encoding: utf-8 -*-
from abjad.tools import abctools
from abjad.tools import timespantools

class TimespanCollection(abctools.AbjadObject):
    r'''A mutable always-sorted collection of timespans.
    ::
    >>> import consort
    >>> timespans = (  
    ...     timespantools.Timespan(0, 3),  
    ...     timespantools.Timespan(1, 3),  
    ...     timespantools.Timespan(1, 2),  
    ...     timespantools.Timespan(2, 5),  
    ...     timespantools.Timespan(6, 9),  
    ...     )  
    >>> timespan_collection = consort.TimespanCollection(timespans)
    '''

    ### CLASS VARIABLES ###
    __slots__ = ('_root_node',)

    ### INITIALIZER ###
    def __init__(self, timespans=None):
        self._root_node = None
        if timespans is not None and timespans:
            self.insert(timespans)

    ### SPECIAL METHODS ###
    def __contains__(self, timespan):
        r'''Is true if this timespan collection contains 'timespan'. Otherwise false.
        ::
        >>> timespans = (  
        ...     timespantools.Timespan(0, 3),
```
... timespantools.Timespan(1, 3),
... timespantools.Timespan(1, 2),
... timespantools.Timespan(2, 5),
... timespantools.Timespan(6, 9),
...)

>>> timespan_collection = consort.TimespanCollection(timespans)

::

>>> timespans[0] in timespan_collection
True

::

>>> timespantools.Timespan(-1, 100) in timespan_collection
False

Returns boolean.

'''

assert TimespanCollection._is_timespan(timespan)
candidates = self.find_timespans_starting_at(timespan.start_offset)
result = timespan in candidates
return result

def __getitem__(self, i):
    r'''Gets timespan at index ‘i’.

    ::

    >>> timespans = (  
    ...     timespantools.Timespan(0, 3),  
    ...     timespantools.Timespan(1, 3),  
    ...     timespantools.Timespan(1, 2),  
    ...     timespantools.Timespan(2, 5),  
    ...     timespantools.Timespan(6, 9),
    ... )
    >>> timespan_collection = consort.TimespanCollection(timespans)

    ::

    >>> timespan_collection[-1]
    Timespan(start_offset=Offset(6, 1), stop_offset=Offset(9, 1))

    ::

    >>> for timespan in timespan_collection[::3]:
    ...     timespan
    ...
    Timespan(start_offset=Offset(0, 1), stop_offset=Offset(3, 1))
    Timespan(start_offset=Offset(1, 1), stop_offset=Offset(2, 1))
    Timespan(start_offset=Offset(1, 1), stop_offset=Offset(3, 1))

    Returns timespan or timespans.

    '''
def recurse_by_index(node, index):
    if node.node_start_index <= index < node.node_stop_index:
        return node.payload[index - node.node_start_index]
    elif node.left_child and index < node.node_start_index:
        return recurse_by_index(node.left_child, index)
    elif node.right_child and node.node_stop_index <= index:
        return recurse_by_index(node.right_child, index)

def recurse_by_slice(node, start, stop):
    result = []
    if node is None:
        return result
    if start < node.node_start_index and node.left_child:
        result.extend(recurse_by_slice(node.left_child, start, stop))
    if start < node.node_stop_index and node.node_start_index < stop:
        node_start = start - node.node_start_index
        node_stop = stop - node.node_start_index
        result.extend(node.payload[node_start:node_stop])
    if node.node_stop_index <= stop and node.right_child:
        result.extend(recurse_by_slice(node.right_child, start, stop))
    return result

if isinstance(i, int):
    if self._root_node is None:
        raise IndexError
    if i < 0:
        i = self._root_node.subtree_stop_index + i
    if i < 0 or self._root_node.subtree_stop_index <= i:
        raise IndexError
    return recurse_by_index(self._root_node, i)
elif isinstance(i, slice):
    if self._root_node is None:
        return []
    indices = i.indices(self._root_node.subtree_stop_index)
    start, stop = indices[0], indices[1]
    return recurse_by_slice(self._root_node, start, stop)
raise TypeError('Indices must be integers or slices, got {}').format(i))

def __iter__(self):
    r'''Iterates timespans in this timespan collection.
    ::
    >>> timespans = (
    ...     timespantools.Timespan(0, 3),
    ...     timespantools.Timespan(1, 3),
    ...     timespantools.Timespan(1, 2),
    ...     timespantools.Timespan(2, 5),
    ...     timespantools.Timespan(6, 9),
    ... )
    >>> timespan_collection = consort.TimespanCollection(timespans)
>>> for timespan in timespan_collection:
    ...    timespan
    ...    Timespan(start_offset=Offset(0, 1), stop_offset=Offset(3, 1))
    ...    Timespan(start_offset=Offset(1, 1), stop_offset=Offset(2, 1))
    ...    Timespan(start_offset=Offset(1, 1), stop_offset=Offset(3, 1))
    ...    Timespan(start_offset=Offset(2, 1), stop_offset=Offset(5, 1))
    ...    Timespan(start_offset=Offset(6, 1), stop_offset=Offset(9, 1))

Returns generator.

```
def recurse(node):
    if node is not None:
        if node.left_child is not None:
            for timespan in recurse(node.left_child):
                yield timespan
        for timespan in node.payload:
            yield timespan
        if node.right_child is not None:
            for timespan in recurse(node.right_child):
                yield timespan
        return recurse(self._root_node)
```

```
def __len__(self):
    r'''Gets length of this timespan collection.
    ::
    >>> timespans = (
        ...    timespantools.Timespan(0, 3),
        ...    timespantools.Timespan(1, 3),
        ...    timespantools.Timespan(1, 2),
        ...    timespantools.Timespan(2, 5),
        ...    timespantools.Timespan(6, 9),
        ...    )
    >>> timespan_collection = consort.TimespanCollection(timespans)
    ::
    >>> len(timespan_collection)
    5

    Returns integer.
    '''
    if self._root_node is None:
        return 0
    return self._root_node.subtree_stop_index
```

```
def __setitem__(self, i, new):
    r'''Sets timespans at index ‘i’ to ‘new’.
    ::
    >>> timespans = (
        ...    timespantools.Timespan(0, 3),
        ...    timespantools.Timespan(1, 3),
        ...    timespantools.Timespan(1, 2),
        ...    timespantools.Timespan(2, 5),
        ...    timespantools.Timespan(6, 9),
        ...    )
    >>> timespan_collection = consort.TimespanCollection(timespans)
    ::
    >>> len(timespan_collection)
    5

    Returns integer.
    '''
    if self._root_node is None:
        return 0
    return self._root_node.subtree_stop_index
```

```
>>> timespans = (  
...    timespantools.Timespan(0, 3),  
...    timespantools.Timespan(1, 3),  
...    timespantools.Timespan(1, 2),  
...    timespantools.Timespan(2, 5),  
...    timespantools.Timespan(6, 9),  
...  )  
>>> timespan_collection = consort.TimespanCollection(timespans)

::

>>> timespan_collection[:3] = [timespantools.Timespan(100, 200)]

Returns none.

if isinstance(i, (int, slice)):  
    old = self[i]  
    self.remove(old)  
    self.insert(new)  
else:  
    message = 'Indices must be ints or slices, got {}'  
    raise TypeError(message)

def __sub__(self, timespan):
    r'''Delete material that intersects `timespan`:

    ::

    >>> timespan_collection = consort.TimespanCollection([  
    ...      timespantools.Timespan(0, 16),  
    ...      timespantools.Timespan(5, 12),  
    ...      timespantools.Timespan(-2, 8),  
    ...    ])  

    ::

    >>> timespan = timespantools.Timespan(5, 10)  
    >>> result = timespan_collection - timespan  

    ::

    >>> print(format(timespan_collection))  
    consort.tools.TimespanCollection(  
        [  
            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),  
            ),  
        ]
    )
Intersects two timespan collections in-place.

```
Ops in place and returns timespan collection.

intersecting_timespans = self.find_timespans_intersecting_timespan(
timespan)
self.remove(intersecting_timespans)
for intersecting_timespan in intersecting_timespans:
    for x in (intersecting_timespan - timespan):
        self.insert(x)
return self
```

### PRIVATE METHODS ###

```
def _insert_node(self, node, start_offset):
    import consort
    if node is None:
        return consort.TimespanCollectionNode(start_offset)
    if start_offset < node.start_offset:
        node.left_child = self._insert_node(node.left_child, start_offset)
else:
    node.right_child = self._insert_node(node.right_child, start_offset)
    return self._rebalance(node)

def _insert_timespan(self, timespan):
    self._root_node = self._insert_node(
        self._root_node,
        timespan.start_offset,
    )
    node = self._search(self._root_node, timespan.start_offset)
    node.payload.append(timespan)
    node.payload.sort(key=lambda x: x.stop_offset)

@staticmethod
def _is_timespan(expr):
    if hasattr(expr, 'start_offset') and hasattr(expr, 'stop_offset'):
        return True
    return False

def _rebalance(self, node):
    if node is not None:
        if 1 < node.balance:
            if 0 <= node.right_child.balance:
                node = self._rotate_right_right(node)
else:
    node = self._rotate_right_left(node)

elif node.balance < -1:
    if node.left_child.balance <= 0:
        node = self._rotate_left_left(node)
    else:
        node = self._rotate_left_right(node)
    assert -1 <= node.balance <= 1
return node

def _remove_node(self, node, start_offset):
    if node is not None:
        if node.start_offset == start_offset:
            if node.left_child and node.right_child:
                next_node = node.right_child
                while next_node.left_child:
                    next_node = next_node.left_child
                node._start_offset = next_node._start_offset
                node._payload = next_node._payload
                node.right_child = self._remove_node(
                    node.right_child,
                    next_node.start_offset,
                )
            else:
                node = node.left_child or node.right_child
        elif node.start_offset < node.start_offset:
            node.left_child = self._remove_node(
                node.left_child,
                start_offset,
            )
        elif node.start_offset < start_offset:
            node.right_child = self._remove_node(
                node.right_child,
                start_offset,
            )
    return self._rebalance(node)

def _remove_timespan(self, timespan, old_start_offset=None):
    start_offset = timespan.start_offset
    if old_start_offset is not None:
        start_offset = old_start_offset
    node = self._search(self._root_node, start_offset)
    if node is None:
        return
    if timespan in node.payload:
        node.payload.remove(timespan)
    elif not node.payload:
        self._root_node = self._remove_node(
            self._root_node,
            start_offset,
        )
    if isinstance(timespan, TimespanCollection):
        timespan._parents.remove(self)
def _rotate_left_left(self, node):
    next_node = node.left_child
    node.left_child = next_node.right_child
    next_node.right_child = node
    return next_node

def _rotate_left_right(self, node):
    node.left_child = self._rotate_right_right(node.left_child)
    next_node = self._rotate_left_left(node)
    return next_node

def _rotate_right_left(self, node):
    node.right_child = self._rotate_left_left(node.right_child)
    next_node = self._rotate_right_right(node)
    return next_node

def _rotate_right_right(self, node):
    next_node = node.right_child
    node.right_child = next_node.left_child
    next_node.left_child = node
    return next_node

def _search(self, node, start_offset):
    if node is not None:
        if node.start_offset == start_offset:
            return node
        elif node.left_child and start_offset < node.start_offset:
            return self._search(node.left_child, start_offset)
        elif node.right_child and node.start_offset < start_offset:
            return self._search(node.right_child, start_offset)
    return None

def _update_indices(self, node):
    def recurse(node, parent_stop_index=None):
        if node is None:
            return
        if node.left_child is not None:
            recurse(node.left_child, parent_stop_index=parent_stop_index)
            node._node_start_index = node.left_child.subtree_stop_index
            node._subtree_start_index = node.left_child.subtree_start_index
        elif parent_stop_index is None:
            node._node_start_index = 0
            node._subtree_start_index = 0
        else:
            node._node_start_index = parent_stop_index
node._subtree_start_index = parent_stop_index
node._node_stop_index = node.node_start_index + len(node.payload)
node._subtree_stop_index = node.node_stop_index
if node.right_child is not None:
    recurse(
        node.right_child,
        parent_stop_index=node.node_stop_index,
    )
node._subtree_stop_index = node.right_child.subtree_stop_index
recurse(node)

def _update_offsets(
    self, node,
):
    if node is None:
        return
    stop_offset_low = min(x.stop_offset for x in node.payload)
    stop_offset_high = max(x.stop_offset for x in node.payload)
    if node.left_child:
        left_child = self._update_offsets(
            node.left_child,
        )
        if left_child.stop_offset_low < stop_offset_low:
            stop_offset_low = left_child.stop_offset_low
        if stop_offset_high < left_child.stop_offset_high:
            stop_offset_high = left_child.stop_offset_high
    if node.right_child:
        right_child = self._update_offsets(
            node.right_child,
        )
        if right_child.stop_offset_low < stop_offset_low:
            stop_offset_low = right_child.stop_offset_low
        if stop_offset_high < right_child.stop_offset_high:
            stop_offset_high = right_child.stop_offset_high
    node._stop_offset_low = stop_offset_low
    node._stop_offset_high = stop_offset_high
    return node

### PRIVATE PROPERTIES ###

@property
def _storage_format_specification(self):
    from abjad.tools import systemtools
    positional_argument_values = ()
    timespans = [x for x in self]
    if timespans:
        positional_argument_values = (timespans,)
    keyword_argument_names = ()
    return systemtools.StorageFormatSpecification(
        self,
        keyword_argument_names=keyword_argument_names,
        positional_argument_values=positional_argument_values,
    )
### PUBLIC METHODS ###

```python
def find_timespans_starting_at(self, offset):
    results = []
    node = self._search(self._root_node, offset)
    if node is not None:
        results.extend(node.payload)
    return tuple(results)
```

```python
def find_timespans_stopping_at(self, offset):
    def recurse(node, offset):
        result = []
        if node is not None:
            if node.stop_offset_low <= offset <= node.stop_offset_high:
                for timespan in node.payload:
                    result.append(timespan)
            if node.left_child is not None:
                result.extend(recurse(node.left_child, offset))
            if node.right_child is not None:
                result.extend(recurse(node.right_child, offset))
        return result
    results = recurse(self._root_node, offset)
    results.sort(key=lambda x: (x.start_offset, x.stop_offset))
    return tuple(results)
```

```python
def find_timespans_overlapping_offset(self, offset):
    r'''Finds timespans overlapping 'offset'.

    >>> timespans = (
    ...     timespantools.Timespan(0, 3),
    ...     timespantools.Timespan(1, 3),
    ...     timespantools.Timespan(1, 2),
    ...     timespantools.Timespan(2, 5),
    ...     timespantools.Timespan(6, 9),
    ... )
    >>> timespan_collection = consort.TimespanCollection(timespans)

    >>> for x in timespan_collection.find_timespans_overlapping_offset(1.5):
    ...     x
    ...
    Timespan(start_offset=Offset(0, 1), stop_offset=Offset(3, 1))
    Timespan(start_offset=Offset(1, 1), stop_offset=Offset(2, 1))
    Timespan(start_offset=Offset(1, 1), stop_offset=Offset(3, 1))
    Returns tuple of 0 or more timespans.
    '''
    def recurse(node, offset, indent=0):
        result = []
```
if node is not None:
    if node.start_offset < offset < node.stop_offset_high:
        result.extend(recurse(node.left_child, offset, indent + 1))
        for timespan in node.payload:
            if offset < timespan.stop_offset:
                result.append(timespan)
        result.extend(recurse(node.right_child, offset, indent + 1))
    elif offset <= node.start_offset:
        result.extend(recurse(node.left_child, offset, indent + 1))
return result
results = recurse(self._root_node, offset)
results.sort(key=lambda x: (x.start_offset, x.stop_offset))
return tuple(results)
def find_timespans_intersecting_timespan(self, timespan):
    r''''Finds timespans overlapping `timespan`.
    ::
        >>> timespans = (
        ...    timespantools.Timespan(0, 3),
        ...    timespantools.Timespan(1, 3),
        ...    timespantools.Timespan(1, 2),
        ...    timespantools.Timespan(2, 5),
        ...    timespantools.Timespan(6, 9),
        ... )
        >>> timespan_collection = consort.TimespanCollection(timespans)
        ::
            >>> timespan = timespantools.Timespan(2, 4)
            >>> for x in timespan_collection.find_timespans_intersecting_timespan(timespan):
            ...    x
            ...    Timespan(start_offset=Offset(0, 1), stop_offset=Offset(3, 1))
            Timespan(start_offset=Offset(1, 1), stop_offset=Offset(3, 1))
            Timespan(start_offset=Offset(2, 1), stop_offset=Offset(5, 1))
        Returns tuple of 0 or more timespans.
    '''
    def recurse(node, timespan):
        result = []
        if node is not None:
            if timespan.intersects_timespan(node):
                result.extend(recurse(node.left_child, timespan))
                for candidate_timespan in node.payload:
                    if candidate_timespan.intersects_timespan(timespan):
                        result.append(candidate_timespan)
                result.extend(recurse(node.right_child, timespan))
            elif (timespan.start_offset <= node.start_offset) or 
                (timespan.stop_offset <= node.start_offset):
                result.extend(recurse(node.left_child, timespan))
        return result
    results = recurse(self._root_node, timespan)
results.sort(key=lambda x: (x.start_offset, x.stop_offset))
return tuple(results)

def get_simultaneity_at(self, offset):
    r'''Gets simultaneity at 'offset'.
    ::
    >>> timespans = {
        ...     timespantools.Timespan(0, 3),
        ...     timespantools.Timespan(1, 3),
        ...     timespantools.Timespan(1, 2),
        ...     timespantools.Timespan(2, 5),
        ...     timespantools.Timespan(6, 9),
        ...     }
    >>> timespan_collection = consort.TimespanCollection(timespans)
    ::
    >>> timespan_collection.get_simultaneity_at(1)
    <TimespanSimultaneity(1 <<3>>)>
    ::
    >>> timespan_collection.get_simultaneity_at(6.5)
    <TimespanSimultaneity(6.5 <<1>>>)
    ...'''
    import consort
    start_timespans = self.find_timespans_starting_at(offset)
    stop_timespans = self.find_timespans_stopping_at(offset)
    overlap_timespans = self.find_timespans_overlapping_offset(offset)
    simultaneity = consort.TimespanSimultaneity(
        timespan_collection=self,
        overlap_timespans=overlap_timespans,
        start_timespans=start_timespans,
        start_offset=offset,
        stop_timespans=stop_timespans,
    )
    return simultaneity

def get_start_offset_after(self, offset):
    r'''Gets start offset in this timespan collection after 'offset'.
    ::
    >>> timespans = {
        ...     timespantools.Timespan(0, 3),
        ...     timespantools.Timespan(1, 3),
        ...     timespantools.Timespan(1, 2),
        ...     timespantools.Timespan(2, 5),
        ...     timespantools.Timespan(6, 9),
        ...     }
    >>> timespan_collection = consort.TimespanCollection(timespans)

713
```python
def recurse(node, offset):
    if node is None:
        return None
    result = None
    if node.start_offset <= offset and node.right_child:
        result = recurse(node.right_child, offset)
    elif offset < node.start_offset:
        result = recurse(node.left_child, offset) or node
    return result
    return recurse(self._root_node, offset)
if result is None:
    return None
return result.start_offset

def get_start_offset_before(self, offset):
    r'''Gets start offset in this timespan collection before 'offset'.

    ::
    >>> timespans = (  
    ...     timespantools.Timespan(0, 3),  
    ...     timespantools.Timespan(1, 3),  
    ...     timespantools.Timespan(1, 2),  
    ...     timespantools.Timespan(2, 5),  
    ...     timespantools.Timespan(6, 9),  
    ...     )  
    >>> timespan_collection = consort.TimespanCollection(timespans)
    >>> timespan_collection.get_start_offset_before(0)
    Offset(0, 1)
    >>> timespan_collection.get_start_offset_before(1)
    Offset(1, 1)
    >>> timespan_collection.get_start_offset_before(2)
    Offset(2, 1)
    >>> timespan_collection.get_start_offset_before(6) is None
    True
    >>> timespan_collection.get_start_offset_before(3)
    Offset(1, 1)
```
```python
def recurse(node, offset):
    if node is None:
        return None
    result = None
    if node.start_offset <= offset:
        result = recurse(node.right_child, offset) or node
    elif offset == node.start_offset or node.left_child:
        result = recurse(node.left_child, offset)
    return result

def index(self, timespan):
    assert self._is_timespan(timespan)
    node = self._search(self._root_node, timespan.start_offset)
    if node is None or timespan not in node.payload:
        raise ValueError('{} not in timespan collection.'.format(timespan))
    index = node.payload.index(timespan) + node.node_start_index
    return index

def insert(self, timespans):
    r'''Inserts 'timespans' into this timespan collection.

    ...
>>> timespan_collection = consort.TimespanCollection()
>>> timespan_collection.insert(timespantools.Timespan(1, 3))
>>> timespan_collection.insert((
... timespantools.Timespan(0, 4),
... timespantools.Timespan(2, 6),
... ))

```
>>> for x in timespan_collection:
...    x
...    ...
...    Timespan(start_offset=Offset(0, 1), stop_offset=Offset(4, 1))
...    Timespan(start_offset=Offset(1, 1), stop_offset=Offset(3, 1))
...    Timespan(start_offset=Offset(2, 1), stop_offset=Offset(6, 1))

'timespans' may be a single timespan or an iterable of timespans.

Returns none.

'''

if self._is_timespan(timespans):
    timespans = [timespans]

for timespan in timespans:
    if not self._is_timespan(timespan):
        continue

    self._insert_timespan(timespan)

    self._update_indices(self._root_node)

    self._update_offsets(self._root_node)

def iterate_simultaneities(
    self,
    reverse=False,
):
    r'''Iterates simultaneities in this timespan collection.

    :
    >>> timespans = (  
    ...    timespantools.Timespan(0, 3),
    ...    timespantools.Timespan(1, 3),
    ...    timespantools.Timespan(1, 2),
    ...    timespantools.Timespan(2, 5),
    ...    timespantools.Timespan(6, 9),
    ...  )
    >>> timespan_collection = consort.TimespanCollection(timespans)
    >>>
    >>> for x in timespan_collection.iterate_simultaneities():
    ...    x
    ...    <TimespanSimultaneity(0 <<1>>)>
    ...    <TimespanSimultaneity(1 <<3>>)>
    ...    <TimespanSimultaneity(2 <<3>>)>

    716
"TimespanSimultaneity(6 <<1>>)
::

>>> for x in timespan_collection.iterate_simultaneities(
...   reverse=True):
...   x
...<TimespanSimultaneity(6 <<1>>)>
<TimespanSimultaneity(2 <<3>>)>
<TimespanSimultaneity(1 <<3>>)>
<TimespanSimultaneity(0 <<1>>)>

Returns generator.

if reverse:
    start_offset = self.latest_start_offset
    simultaneity = self.get_simultaneity_at(start_offset)
    yield simultaneity
    simultaneity = simultaneity.previous_simultaneity
    while simultaneity is not None:
        yield simultaneity
        simultaneity = simultaneity.previous_simultaneity
else:
    start_offset = self.earliest_start_offset
    simultaneity = self.get_simultaneity_at(start_offset)
    yield simultaneity
    simultaneity = simultaneity.next_simultaneity
    while simultaneity is not None:
        yield simultaneity
        simultaneity = simultaneity.next_simultaneity

def iterate_simultaneities_nwise(
    self,
    n=3,
    reverse=False,
)::
    r"""Iterates simultaneities in this timespan collection in groups of 'n'.
::

>>> timespans = (
...   ... timespantools.Timespan(0, 3),
...   ... timespantools.Timespan(1, 3),
...   ... timespantools.Timespan(1, 2),
...   ... timespantools.Timespan(2, 5),
...   ... timespantools.Timespan(6, 9),
...   ... )
>>> timespan_collection = consort.TimespanCollection(timespans)
::
>>> for x in timespan_collection.iterate_simultaneities_nwise(n=2):
...   x
...   ...

(<TimespanSimultaneity(0 <<1>>, TimespanSimultaneity(1 <<3>>))>,
(<TimespanSimultaneity(1 <<3>>, TimespanSimultaneity(2 <<3>>))>,
(<TimespanSimultaneity(2 <<3>>, TimespanSimultaneity(6 <<1>>))>

::

>>> for x in timespan_collection.iterate_simultaneities_nwise(n=2, reverse=True):
...   x
...

(<TimespanSimultaneity(2 <<3>>, TimespanSimultaneity(6 <<1>>))>,
(<TimespanSimultaneity(1 <<3>>, TimespanSimultaneity(2 <<3>>))>,
(<TimespanSimultaneity(0 <<1>>, TimespanSimultaneity(1 <<3>>)))

Returns generator.

2nd

n = int(n)
assert 0 < n
if reverse:
  for simultaneity in self.iterate_simultaneities(reverse=True):
    simultaneities = [simultaneity]
    while len(simultaneities) < n:
      next_simultaneity = simultaneities[-1].next_simultaneity
      if next_simultaneity is None:
        break
      simultaneities.append(next_simultaneity)
    if len(simultaneities) == n:
      yield tuple(simultaneities)
else:
  for simultaneity in self.iterate_simultaneities():
    simultaneities = [simultaneity]
    while len(simultaneities) < n:
      previous_simultaneity = simultaneities[-1].previous_simultaneity
      if previous_simultaneity is None:
        break
      simultaneities.append(previous_simultaneity)
    if len(simultaneities) == n:
      yield tuple(reversed(simultaneities))

def remove(self, timespans):
  r'''Removes timespans from this timespan collection.

::

>>> timespans = (
...   timespantools.Timespan(0, 3),
...   timespantools.Timespan(1, 3),
...   timespantools.Timespan(1, 2),
...   timespantools.Timespan(2, 5),
...   timespantools.Timespan(6, 9),
... )
>>> timespan_collection = consort.TimespanCollection(timespans)
::

>>> timespan_collection.remove(timespans[1:-1])
::

>>> for timespan in timespan_collection:
...    timespan
...    Timespan(start_offset=Offset(0, 1), stop_offset=Offset(3, 1))
...    Timespan(start_offset=Offset(6, 1), stop_offset=Offset(9, 1))
...

```

if self._is_timespan(timespans):
    timespans = [timespans]
for timespan in timespans:
    if not self._is_timespan(timespan):
        continue
    self._remove_timespan(timespan)
    self._update_indices(self._root_node)
    self._update_offsets(self._root_node)
```

### PUBLIC PROPERTIES ###

@property
def all_offsets(self):
    offsets = set()
    for timespan in self:
        offsets.add(timespan.start_offset)
        offsets.add(timespan.stop_offset)
    return tuple(sorted(offsets))

@property
def all_start_offsets(self):
    start_offsets = set()
    for timespan in self:
        start_offsets.add(timespan.start_offset)
    return tuple(sorted(start_offsets))

@property
def all_stop_offsets(self):
    stop_offsets = set()
    for timespan in self:
        stop_offsets.add(timespan.stop_offset)
    return tuple(sorted(stop_offsets))

@property
def earliest_start_offset(self):
    def recurse(node):
        if node.left_child is not None:
            return recurse(node.left_child)
    return node.start_offset
if self._root_node is not None:
    return recurse(self._root_node)
return float('-inf')

@property
def earliest_stop_offset(self):
    if self._root_node is not None:
        return self._root_node.stop_offset_low
    return float('inf')

@property
def latest_start_offset(self):
    def recurse(node):
        if node.right_child is not None:
            return recurse(node._right_child)
        return node.start_offset
    if self._root_node is not None:
        return recurse(self._root_node)
    return float('-inf')

@property
def latest_stop_offset(self):
    if self._root_node is not None:
        return self._root_node.stop_offset_high
    return float('inf')

@property
def start_offset(self):
    return self.earliest_start_offset

@property
def stop_offset(self):
    return self.latest_stop_offset

A.55  consort.tools.TimespanCollectionNode

class TimespanCollectionNode(abctools.AbjadObject):
    r'''A node in a timespan collection.
    '''

    __slots__ = ('_balance',
                 '_height',
                 '_left_child',
                 '_node_start_index',
                 '_node_stop_index',
                 '_payload',
```python
'right_child',
'_start_offset',
'_stop_offset_high',
'_stop_offset_low',
'_subtree_start_index',
'_subtree_stop_index',
)

### INITIALIZER ###

def __init__(self, start_offset=0):
    self._balance = 0
    self._height = 0
    self._left_child = None
    self._node_start_index = -1
    self._node_stop_index = -1
    self._payload = []
    self._right_child = None
    self._start_offset = start_offset
    self._stop_offset_high = None
    self._stop_offset_low = None
    self._subtree_start_index = -1
    self._subtree_stop_index = -1

### SPECIAL METHODS ###

def __repr__(self):
    r'''Gets the repr of this timespan collection node.
    '''
    return '<Node: Start:{} Indices:({}:{}:{}) Length:{{}}}>
        .format(self.start_offset,
                self.subtree_start_index,
                self.node_start_index,
                self.node_stop_index,
                self.subtree_stop_index,
                len(self.payload),
        )

### PRIVATE METHODS ###

def _debug(self):
    return '
    .join(self._get_debug_pieces())

def _get_debug_pieces(self):
    result = []
    result.append(repr(self))
    if self.left_child:
        subresult = self.left_child._get_debug_pieces()
        result.append('  L: {}'.format(subresult[0]))
        result.extend('    ' + x for x in subresult[1:])
    if self.right_child:
        subresult = self.right_child._get_debug_pieces()
        result.append('  R: {}'.format(subresult[0]))
        result.extend('    ' + x for x in subresult[1:])
```
    return result

    def _update(self):
        left_height = -1
        right_height = -1
        if self.left_child is not None:
            left_height = self.left_child.height
        if self.right_child is not None:
            right_height = self.right_child.height
        self._height = max(left_height, right_height) + 1
        self._balance = right_height - left_height
        return self.height

    ### PUBLIC PROPERTIES ###

    @property
    def balance(self):
        '''Gets the balance of this timespan collection node.
        '''
        return self._balance

    @property
    def height(self):
        '''Gets the height of this timespan collection node.
        '''
        return self._height

    @property
    def left_child(self):
        '''Gets and sets the left child of this timespan collection node.
        '''
        return self._left_child

    @left_child.setter
    def left_child(self, node):
        self._left_child = node
        self._update()

    @property
    def node_start_index(self):
        '''Gets the node start index of this timespan collection node.
        '''
        return self._node_start_index

    @property
    def node_stop_index(self):
        '''Gets the node stop index of this timespan collection node.
        '''
        return self._node_stop_index

    @property
    def payload(self):
        '''Gets the payload of this timespan collection node.
        '''

return self._payload

@property
def right_child(self):
    '''Gets and sets the right child of this timespan collection node.
    '''
    return self._right_child

@right_child.setter
def right_child(self, node):
    self._right_child = node
    self._update()

@property
def start_offset(self):
    '''Gets the start offset of this timespan collection node.
    '''
    return self._start_offset

@property
def stop_offset_high(self):
    '''Gets the highest stop offset of the subtree rooted on this timespan collection node.
    '''
    return self._stop_offset_high

@property
def stop_offset_low(self):
    '''Gets the lowest stop offset of the subtree rooted on this timespan collection node.
    '''
    return self._stop_offset_low

@property
def subtree_start_index(self):
    '''Gets the start index of the subtree rooted on this timespan collection node.
    '''
    return self._subtree_start_index

@property
def subtree_stop_index(self):
    '''Gets the stop index of the subtree rooted on this timespan collection node.
    '''
    return self._subtree_stop_index

@property
def timespan(self):
    return timespantools.Timespan(
        start_offset=self.start_offset,
        stop_offset=self.stop_offset_high,
    )
A.56 consort.tools.TimespanInventoryMapping

```python
# -*- encoding: utf-8 -*-
from abjad.tools import timespantools

class TimespanInventoryMapping(dict):

    ### SPECIAL METHODS ###

    def __illustrate__(self, range_=None, scale=None):
        timespan_inventory = timespantools.TimespanInventory()
        for key, value in self.items():
            timespan_inventory.extend(value)
        return timespan_inventory.__illustrate__(
            key='voice_name',
            range_=range_,
            scale=scale,
        )
```

A.57 consort.tools.TimespanMaker

```python
# -*- encoding: utf-8 -*-
from __future__ import print_function
import abc
import collections
from abjad import new
from abjad.tools import abctools
from abjad.tools import durationtools
from abjad.tools import rhythmtools
from abjad.tools import timespantools

class TimespanMaker(abctools.AbjadValueObject):

    r'''Abstract base class for timespan makers.

    ..  container:: special

        Example:

        ..  container::

            >>> import timespantools
            >>> timespan_inventory = timespantools.TimespanInventory()
            >>> for key, value in self.items():
            >>>     timespan_inventory.extend(value)
            >>> return timespan_inventory.__illustrate__(
            >>>     key='voice_name',
            >>>     range_=range_,
            >>>     scale=scale,
            >>> )
```
```
timespan_specifier=None,
):
import consort
if output_masks is not None:
    if isinstance(output_masks, rhythmmapertools.BooleanPattern):
        output_masks = rhythmmapertools.BooleanPatternInventory(
            items=output_masks,
        )
self._output_masks = output_masks
if padding is not None:
    padding = durationtools.Duration(padding)
self._padding = padding
if seed is not None:
    seed = int(seed)
self._seed = seed
if timespan_specifier is not None:
    assert isinstance(timespan_specifier, consort.TimespanSpecifier)
self._timespan_specifier = timespan_specifier

### SPECIAL METHODS ###

def __call__(
    self,
    layer=None,
    music_specifiers=None,
    rotation=None,
    silenced_context_names=None,
    target_timespan=None,
    timespan_inventory=None,
):
    if not isinstance(timespan_inventory, timespantools.TimespanInventory):
        timespan_inventory = timespantools.TimespanInventory(
            timespan_inventory,
        )
    if target_timespan is None:
        if timespan_inventory:
            target_timespan = timespan_inventory.timespan
        else:
            raise TypeError
    assert isinstance(timespan_inventory, timespantools.TimespanInventory)
    if not music_specifiers:
        return timespan_inventory
    music_specifiers = self._coerce_music_specifiers(music_specifiers)
    new_timespans = self._make_timespans(
        layer=layer,
        music_specifiers=music_specifiers,
        target_timespan=target_timespan,
        timespan_inventory=timespan_inventory,
    )
    self._cleanup_silent_timespans(
        layer=layer,
        silenced_context_names=silenced_context_names,
        timespans=new_timespans,
```python
    timespan_inventory.extend(new_timespans)
timespan_inventory.sort()
return timespan_inventory

### PRIVATE METHODS ###

@staticmethod
def _coerce_music_specifiers(music_specifiers):
    import consort
    result = collections.OrderedDict()
    prototype = (consort.MusicSpecifierSequence,
                 consort.CompositeMusicSpecifier,
                 )
    for context_name, music_specifier in music_specifiers.items():
        if music_specifier is None:
            music_specifier = [None]
        if not isinstance(music_specifier, prototype):
            music_specifier = consort.MusicSpecifierSequence(music_specifiers=music_specifier,
                                                             )
        result[context_name] = music_specifier
    return result

def _cleanup_silent_timespans(self, layer, silenced_context_names, timespans,
)::
    import consort
    if not silenced_context_names or not timespans:
        return

    silent_timespans_by_context = {}
    for context_name in silenced_context_names:
        if context_name not in silent_timespans_by_context:
            silent_timespans_by_context[context_name] = \
                timespantools.TimespanInventory()

    sounding_timespans_by_context = {}
    sounding_timespans = timespantools.TimespanInventory()

    for timespan in timespans:
        voice_name = timespan.voice_name
        if isinstance(timespan, consort.PerformedTimespan):
            if voice_name not in sounding_timespans_by_context:
                sounding_timespans_by_context[voice_name] = \
                    timespantools.TimespanInventory()
            sounding_timespans_by_context[voice_name].append(timespan)
            sounding_timespans.append(timespan)
        else:
            if voice_name not in silent_timespans_by_context:
```
silent_timespans_by_context[voice_name] = \
    timespantools.TimepanInventory()
    silent_timespans_by_context[voice_name].append(timespan)

sounding_timespans.sort()
sounding_timespans.compute_logical_or()

# Create silences.
for shard in sounding_timespans.partition(True):
    for context_name in silenced_context_names:
        timespan = consort.SilentTimespan(
            layer=layer,
            voice_name=context_name,
            start_offset=shard.start_offset,
            stop_offset=shard.stop_offset,
        )
        silent_timespans_by_context[context_name].append(timespan)

# Remove any overlap between performed and silent timespans.
# Then add the silent timespans into the original timespan inventory.
for context_name, silent_timespans in \
    sorted(silent_timespans_by_context.items()):
    silent_timespans.sort()
    if context_name in sounding_timespans_by_context:
        for timespan in sounding_timespans_by_context[context_name]:
            silent_timespans - timespan
        timespans.extend(silent_timespans)

### PUBLIC METHODS ###

def rotate(self, rotation):
    seed = self.seed or 0
    seed = seed + rotation
    return new(self, seed=seed)

### PUBLIC PROPERTIES ###

@property
def is_dependent(self):
    return False

@property
def output_masks(self):
    return self._output_masks

@property
def padding(self):
    return self._padding

@property
def seed(self):
    return self._seed

@property
def timespan_specifier(self):
    return self._timespan_specifier

A.58   consort.tools.TimespanSimultaneity

# -*- encoding: utf-8 -*-
from abjad import abctools

class TimespanSimultaneity(abctools.AbjadObject):
    r'''A simultaneity of timespans in a timespan collection.'''

    ### CLASS VARIABLES ###

    __slots__ = ('_timespan_collection',
                 '_overlap_timespans',
                 '_start_timespans',
                 '_start_offset',
                 '_stop_timespans',
                 )

    ### INITIALIZER ###

    def __init__(self,
                 timespan_collection=None,
                 overlap_timespans=None,
                 start_timespans=None,
                 start_offset=None,
                 stop_timespans=None,
                 ):
        import consort
        if timespan_collection is not None:
            prototype = consort.TimespanCollection
            assert isinstance(timespan_collection, prototype)
            self._timespan_collection = timespan_collection
            self._start_offset = start_offset
            assert isinstance(start_timespans, (tuple, type(None)))
            assert isinstance(stop_timespans, (tuple, type(None)))
            assert isinstance(overlap_timespans, (tuple, type(None)))
            self._start_timespans = start_timespans
            self._stop_timespans = stop_timespans
            self._overlap_timespans = overlap_timespans

    ### SPECIAL METHODS ###

    def __repr__(self):
        r'''Gets the repr of this simultaneity.'''
        return '<{}({} <<{}>>)'.format(
            type(self).__name__,
            str(self.start_offset),
            }
len(self.start_timespans) + len(self.overlap_timespans),
)

### PUBLIC PROPERTIES ###

@property
def next_simultaneity(self):
    r'''Gets the next simultaneity in this simultaneity's timespan collection.
    '''
    tree = self._timespan_collection
    if tree is None:
        return None
    start_offset = tree.get_start_offset_after(self.start_offset)
    if start_offset is None:
        return None
    return tree.get_simultaneity_at(start_offset)

@property
def next_start_offset(self):
    r'''Gets the next simultaneity start offset in this simultaneity's timespan collection.
    '''
    tree = self._timespan_collection
    if tree is None:
        return None
    start_offset = tree.get_start_offset_after(self.start_offset)
    return start_offset

@property
def overlap_timespans(self):
    r'''Gets the timespans in this simultaneity which overlap this simultaneity's start offset.
    '''
    return self._overlap_timespans

@property
def previous_simultaneity(self):
    r'''Gets the previous simultaneity in this simultaneity's timespan collection.
    '''
    tree = self._timespan_collection
    if tree is None:
        return None
    start_offset = tree.get_start_offset_before(self.start_offset)
    if start_offset is None:
        return None
    return tree.get_simultaneity_at(start_offset)

@property
def previous_start_offset(self):
    r'''Gets the previous simultaneity start offset in this simultaneity's timespan collection.
    '''
    }
tree = self._timespan_collection
if tree is None:
    return None
start_offset = tree.get_start_offset_before(self.start_offset)
return start_offset

@property
def start_offset(self):
    r'''Gets this simultaneity's start offset.
    '''
    return self._start_offset

@property
def start_timespans(self):
    r'''Gets the timespans in this simultaneity which start at this
    simultaneity's start offset.
    '''
    return self._start_timespans

@property
def stop_timespans(self):
    r'''Gets the timespans in this simultaneity which stop at this
    simultaneity's start offset.
    '''
    return self._stop_timespans

@property
def timespan_collection(self):
    r'''Gets this simultaneity's timespan collection.
    '''
    return self._timespan_collection

### INITIALIZER ###
def __init__(self,
            forbid_fusing=None,
forbid_splitting=None,
minimum_duration=None,
):
    if forbid_fusing is not None:
        forbid_fusing = bool(forbid_fusing)
    self._forbid_fusing = forbid_fusing
    if forbid_splitting is not None:
        forbid_splitting = bool(forbid_splitting)
    self._forbid_splitting = forbid_splitting
    if minimum_duration is not None:
        minimum_duration = durationtools.Duration(minimum_duration)
    self._minimum_duration = minimum_duration

### PUBLIC PROPERTIES ###

@property
def forbid_fusing(self):
    return self._forbid_fusing

@property
def forbid_splitting(self):
    return self._forbid_splitting

@property
def minimum_duration(self):
    return self._minimum_duration
This page intentionally left blank.
B.1  zaira

B.1.1  zaira.makers.Percussion

```python
# -*- encoding: utf-8 -*-
from abjad.tools import pitchtools

class Percussion(object):

    BRAKE_DRUM = pitchtools.NamedPitch("g'")
    HIGH_CYMBAL = pitchtools.NamedPitch("e'")
    MIDDLE_CYMBAL = pitchtools.NamedPitch("c'")
    LOW_CYMBAL = pitchtools.NamedPitch("a")
    TAM_TAM = pitchtools.NamedPitch("f")
    TAMBOURINE = pitchtools.NamedPitch("e'")
    GUERO = pitchtools.NamedPitch("c'")
    BAMBOO_WINDCHIMES = pitchtools.NamedPitch("a")
    HIGH_TOM = pitchtools.NamedPitch("f'")
    LOW_TOM = pitchtools.NamedPitch("d'")
    KICK_DRUM = pitchtools.NamedPitch("b")
    BASS_DRUM = pitchtools.NamedPitch("g")
```

B.1.2  zaira.makers.ZairaScoreTemplate

```python
# -*- encoding: utf-8 -*-
import collections
from abjad.tools import abctools
from abjad.tools import indicatortools
from abjad.tools import instrumenttools
from abjad.tools import scoretools
```
```python
import consort

class ZairaScoreTemplate(abctools.AbjadValueObject):
    r'''Zaira score template.

>>> import zaira
>>> template = zaira.makers.ZairaScoreTemplate()
>>> score = template()
>>> print(format(score))
\context Score = "Zaira Score" <<
  \tag #'time
  \context TimeSignatureContext = "Time Signature Context" { }
\context EnsembleGroup = "Wind Section Staff Group" <<
  \tag #'flute
  \context PerformerGroup = "Flute Performer Group" \with {
      instrumentName = \markup { Flute }
      shortInstrumentName = \markup { Fl. }
  } <<
  \context FluteStaff = "Flute Staff" {
    \context Voice = "Flute Voice" {
      \clef "treble"
    }
  }

>>>
\tag #'oboe
\context PerformerGroup = "Oboe Performer Group" \with {
    instrumentName = \markup { Oboe }
    shortInstrumentName = \markup { Ob. }
} <<
\context OboeStaff = "Oboe Staff" {
    \context Voice = "Oboe Voice" {
      \clef "treble"
    }
}

>>>
\tag #'clarinet-in-b-flat
\context PerformerGroup = "Clarinet In B-Flat Performer Group" \with {
    instrumentName = \markup { Clarinet in B-flat }
    shortInstrumentName = \markup { Cl. in B-flat }
} <<
\context ClarinetInBFlatStaff = "Clarinet In B-Flat Staff" {
    \context Voice = "Clarinet In B-Flat Voice" {
      \clef "treble"
    }
}

>>>
\tag #'percussion
\context EnsembleGroup = "Percussion Section Staff Group" <<
  \context PerformerGroup = "Metals Performer Group" \with {
```
instrumentName = \markup { Metals }  
shortInstrumentName = \markup { Metals } 
} <<  
\context MetalsStaff = "Metals Staff" {  
\context Voice = "Metals Voice" {  
\clef "percussion"  
}  
}  
>>  
\context PerformerGroup = "Woods Performer Group" \with {  
instrumentName = \markup { Woods }  
shortInstrumentName = \markup { Woods }  
} <<  
\context WoodsStaff = "Woods Staff" {  
\context Voice = "Woods Voice" {  
\clef "percussion"  
}  
}  
>>  
\context PerformerGroup = "Drums Performer Group" \with {  
instrumentName = \markup { Drums }  
shortInstrumentName = \markup { Drums }  
} <<  
\context DrumsStaff = "Drums Staff" {  
\context Voice = "Drums Voice" {  
\clef "percussion"  
}  
}  
>>  
\tag #'piano  
\context PianoStaff = "Piano Performer Group" \with {  
instrumentName = \markup { Piano }  
shortInstrumentName = \markup { Pf. }  
} <<  
\context PianoUpperStaff = "Piano Upper Staff" {  
\context Voice = "Piano Upper Voice" {  
\clef "treble"  
}  
\context Dynamics = "Piano Dynamics" {  
}  
\context PianoLowerStaff = "Piano Lower Staff" {  
\context Voice = "Piano Lower Voice" {  
\clef "bass"  
}  
\context Dynamics = "Piano Pedals" {  
}  
}  
>>  
\context EnsembleGroup = "String Section Staff Group" <<  
\tag #'violin  
\context StringPerformerGroup = "Violin Performer Group" \with {  
instrumentName = \markup { Violin }
shortInstrumentName = \markup { Vn. } \<<
\context StringStaff = "Violin Staff" { \context Voice = "Violin Voice" { \clef "treble" } }
\>>
\tag #'viola
\context StringPerformerGroup = "Viola Performer Group" \with { instrumentName = \markup { Viola } shortInstrumentName = \markup { Va. } } \<<
\context StringStaff = "Viola Staff" { \context Voice = "Viola Voice" { \clef "alto" } }
\>>
\tag #'cello
\context StringPerformerGroup = "Cello Performer Group" \with { instrumentName = \markup { Cello } shortInstrumentName = \markup { Vc. } } \<<
\context StringStaff = "Cello Staff" { \context Voice = "Cello Voice" { \clef "bass" } }
\>>
>>
'''

### CLASS VARIABLES ###

__slots__ = ('_context_name_abbreviations',)

### INITIALIZER ###

def __init__(self):
    self._context_name_abbreviations = collections.OrderedDict()

### SPECIAL METHODS ###

def __call__(self):
    manager = consort.ScoreTemplateManager

### WINDS ###
flute = manager.make_single_wind_performer(
    clef=indicatortools.Clef('treble'),
    instrument=instrumenttools.Flute(),
    score_template=self,
)

oboe = manager.make_single_wind_performer(
    clef=indicatortools.Clef('treble'),
    instrument=instrumenttools.Oboe(),
    score_template=self,
)

clarinet = manager.make_single_wind_performer(
    abbreviation='clarinet',
    clef=indicatortools.Clef('treble'),
    instrument=instrumenttools.ClarinetInBFlat(),
    score_template=self,
)

winds = manager.make_ensemble_group(
    name='Wind Section Staff Group',
    performer_groups=[
        flute,
        oboe,
        clarinet,
    ],
)

### PERCUSSION ###

metals = manager.make_single_basic_performer(
    clef=indicatortools.Clef('percussion'),
    instrument=instrumenttools.Percussion(
        instrument_name='Metals',
        short_instrument_name='Metals',
    ),
    score_template=self,
)

woods = manager.make_single_basic_performer(
    clef=indicatortools.Clef('percussion'),
    instrument=instrumenttools.Percussion(
        instrument_name='Woods',
        short_instrument_name='Woods',
    ),
    score_template=self,
)

drums = manager.make_single_basic_performer(
    clef=indicatortools.Clef('percussion'),
    instrument=instrumenttools.Percussion(
        instrument_name='Drums',
        short_instrument_name='Drums',
    ),
)
score_template=self,
)

percussion = manager.make_ensemble_group(
    label='percussion',
    name='Percussion Section Staff Group',
    performer_groups=[
        metals,
        woods,
        drums,
    ],
)

### PIANO ###

piano = manager.make_single_piano_performer(
    instrument=instrumenttools.Piano(),
    score_template=self,
)

### STRINGS ###

violin = manager.make_single_string_performer(
    clef=indicatortools.Clef('treble'),
    instrument=instrumenttools.Violin(),
    split=False,
    score_template=self,
)

viola = manager.make_single_string_performer(
    clef=indicatortools.Clef('alto'),
    instrument=instrumenttools.Viola(),
    split=False,
    score_template=self,
)

cello = manager.make_single_string_performer(
    clef=indicatortools.Clef('bass'),
    instrument=instrumenttools.Cello(),
    split=False,
    score_template=self,
)

strings = manager.make_ensemble_group(
    name='String Section Staff Group',
    performer_groups=[
        violin,
        viola,
        cello,
    ],
)

### SCORE ###
time_signature_context = manager.make_time_signature_context()

score = scoretools.Score(
    [  
        time_signature_context,  
        winds,                 
        percussion,            
        piano,                 
        strings,               
    ],  

    name='Zaira Score',
)

return score  

### PUBLIC PROPERTIES ###

@property
def context_name_abbreviations(self):
    return self._context_name_abbreviations

B.1.3 zaira.makers.ZairaSegmentMaker

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import markuptools

class ZairaSegmentMaker(consort.SegmentMaker):

    ### CLASS VARIABLES ###

    __slots__ = ()

    ### INITIALIZER ###

    def __init__(
        self,
        annotate_colors=None,
        annotate_phrasing=None,
        annotate_timespans=None,
        desired_duration_in_seconds=None,
        discard_final_silence=None,
        maximum_meter_run_length=None,
        name=None,
        omit_stylesheets=None,
        permitted_time_signatures=None,
        repeat=None,
        score_template=None,
        settings=None,
        tempo=None,
        timespan_quantization=None,
    ):
        import zaira

import zaira

739
permitted_time_signatures = permitted_time_signatures or 
    zaira.materials.time_signatures
score_template = score_template or zaira.makers.ZairaScoreTemplate()

consort.SegmentMaker.__init__(
    self,
    annotate_colors=annotate_colors,
    annotate_phrasing=annotate_phrasing,
    annotate_timespans=annotate_timespans,
    desired_duration_in_seconds=desired_duration_in_seconds,
    discard_final_silence=discard_final_silence,
    maximum_meter_run_length=maximum_meter_run_length,
    name=name,
    omit_stylesheets=omit_stylesheets,
    permitted_time_signatures=permitted_time_signatures,
    repeat=repeat,
    score_template=score_template,
    settings=settings,
    tempo=tempo,
    timespan_quantization=timespan_quantization,
)

### PUBLIC PROPERTIES ###

@property
def final_markup(self):
    jamaica_plain = markuptools.Markup('Jamaica Plain, OR')
    queens = markuptools.Markup('Fresh Meadows, NY')
    date = markuptools.Markup('June 2014 - September 2014')
    null = markuptools.Markup.null()
    contents = [
        null,
        null,
        null,
        jamaica_plain,
        queens,
        date,
    ]
    markup = markuptools.Markup.right_column(contents)
    markup = markup.italic()
    return markup

@property
def score_package_name(self):
    return 'zaira'

B.2 ZAIRA MATERIALS

B.2.1 ZAIRA.MATERIALS.BACKGROUND_DYNAMIC_ATTACHMENT_EXPRESSION

# -*- encoding: utf-8 -*-
from abjad import *
import consort
background_dynamic_attachment_expression = consort.AttachmentExpression(
    attachments=(
        consort.SimpleDynamicExpression('ppp'),
        consort.SimpleDynamicExpression('p'),
        consort.SimpleDynamicExpression('pp'),
    ),
    selector=selectortools.select_pitched_runs(),
)

B.2.2  zaira.materials.brazil_nut_music_specifier

brazil_nut_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        clef_spanner=consort.ClefSpanner('percussion'),
        staff_lines_spanner=spannertools.StaffLinesSpanner(
            lines=(4, -4),
            overrides={'note_head__no_ledgers': True,
                       'note_head__style': 'cross',
                       },
        ),
    ),
    staccato=consort.AttachmentExpression(
        attachments=indicatortools.Articulation('..'),
        selector=selectortools.Selector().by_logical_tie(pitched=True).
            .by_duration('<', (1, 8)).by_length(1),
    ),
    stem_tremolo_spanner=consort.AttachmentExpression(
        attachments=spannertools.StemTremoloSpanner(),
        selector=selectortools.Selector().by_logical_tie(pitched=True).
            .by_duration('>', (1, 16)),
    ),
    text_spanner=consort.AttachmentExpression(
        attachments=consort.ComplexTextSpanner(
            markup=Markup(r'\concat { \vstrut shaker }')
                .italic()
                .pad_around(0.5)
                .box(),
        ),
        selector=selectortools.Selector().by_leaves(),
    ),
)
B.2.3 \texttt{zaira.materials.cello\_solo\_music\_specifier}

```python
# -*- encoding: utf-8 -*-

from abjad import Markup
from abjad.tools import indicatortools
from abjad.tools import selectortools
from abjad.tools import spannertools
from abjad.tools.topleveltools import new
import consort

import zaira


cello_solo_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.erratic_dynamic_attachment_expression,
        trill_spanner=consort.AttachmentExpression(
            attachments=(
                None,
                spannertools.ComplexTrillSpanner(interval='+m3'),
                None,
                spannertools.ComplexTrillSpanner(interval='+m3'),
                None,
                spannertools.ComplexTrillSpanner(interval='+M2'),
            ),
        selector=selectortools.Selector()
        .by_leaves()
        .by_logical_tie(pitched=True)
    ),
    tenuto=consort.AttachmentExpression(
        attachments=indicatortools.Articulation('tenuto'),
        selector=selectortools.Selector()
        .by_leaves()
        .by_logical_tie(pitched=True)[0],
    ),
    text_spanner=consort.AttachmentExpression(
        attachments=(
            consort.ComplexTextSpanner(
                markup=Markup(r'\concat { \vstrut "col legno" }')
                .italic()
                .pad_around(0.5)
                .box(),
            None,
            spannertools.Glissando(),
        ),
        selector=selectortools.select_pitched_runs(),
    ),
    rhythm_maker=zaira.materials.undergrowth_rhythm_maker,
)
```
B.2.4 \textsc{Zaira.Materials.Dense\_Timespan\_Maker}

```python
# -*- encoding: utf-8 -*-
from abjad import *
import consort

dense_timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=rhythmmakertools.Talea(
        counts=(0, 3, 4, 2, 5),
        denominator=16,
    ),
    playing_talea=rhythmmakertools.Talea(
        counts=(6, 8, 4, 5, 6, 4),
        denominator=16,
    ),
    playing_groupings=(2, 1, 2, 3, 1, 1, 2, 2),
    repeat=True,
    silence_talea=rhythmmakertools.Talea(
        counts=(2, 4, 6, 3, 4, 10),
        denominator=16,
    ),
    step_anchor=Right,
    synchronize_groupings=False,
    synchronize_step=False,
    timespan_specifier=consort.TimespanSpecifier(
        minimum_duration=durationtools.Duration(1, 8),
    ),
)
```

B.2.5 \textsc{Zaira.Materials.Drum\_Agitation\_Music\_Specifier}

```python
# -*- encoding: utf-8 -*-
from abjad.tools import indicatortools
from abjad.tools import pitchtools
from abjad.tools import selectortools
from abjad.tools.topleveltools import new
import consort
import zaira

drum_agitation_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        pitch_handler=consort.AbsolutePitchHandler(
            pitch_specifier="d, f, d, fqs, ef, d, ef, f, fqs, d, g, d, d, as,",
            pitch_application_rate='division',
        ),
    ),
    rhythm_maker=new(
        zaira.materials.reiterating_rhythm_maker,
        denominators=(8, 4, 8, 1),
    ),
)
```
B.2.6  ZAIRA.MATERIALS.DRUM_BROUSHED_MUSIC_SPECIFIER

dynamic_expression=zaira.materials.foreground_dynamic_attachment_expression,
accent=consort.AttachmentExpression(
    attachments=indicatortools.Articulation('accent'),
    selector=selectortools.select_first_logical_tie_in_pitched_runs()[0],
),
staccato=consort.AttachmentExpression(
    attachments=indicatortools.Articulation('staccato'),
    selector=selectortools.select_all_but_first_logical_tie_in_pitched_runs()[0],
),
pitch_handler=consort.AbsolutePitchHandler(
pitch_specifier=pitchtools.PitchSegment(
    items=(
        zaira.makers.Percussion.HIGH_TOM,
        zaira.makers.Percussion.LOW_TOM,
        zaira.makers.Percussion.BASS_DRUM,
        zaira.makers.Percussion.HIGH_TOM,
        zaira.makers.Percussion.BASS_DRUM,
        zaira.makers.Percussion.LOW_TOM,
        zaira.makers.Percussion.HIGH_TOM,
        zaira.makers.Percussion.BASS_DRUM,
        zaira.makers.Percussion.LOW_TOM,
        ),
    ),
    ),
rhythm_maker=new(
    zaira.materials.stuttering_rhythm_maker,
    extra_counts_per_division=(2, 1, 2, 1, 0, 2, 1, 0),
    )
)
stem_tremolo_spanner = consort.AttachmentExpression(
    attachments=(
        spannertools.StemTremoloSpanner(),
        None,
    ),
    selector=selectortools.select_pitched_runs(),
),

pitch_handler = consort.AbsolutePitchHandler(
pitch_specifier = pitchtools.PitchSegment(
    items=(
        zaira.makers.Percussion.HIGH_TOM,
        zaira.makers.Percussion.LOW_TOM,
        zaira.makers.Percussion.BASS_DRUM,
        zaira.makers.Percussion.HIGH_TOM,
        zaira.makers.Percussion.BASS_DRUM,
        zaira.makers.Percussion.LOW_TOM,
        zaira.makers.Percussion.BASS_DRUM,
        zaira.makers.Percussion.LOW_TOM,
    ),
),

rhythm_maker = zaira.materials.sustained_rhythm_maker,
)

B.2.7 ZAIRA.MATERIALS.DRUM_HEARTBEAT_MUSICSpecifier

# -*- encoding: utf-8 -*-
from abjad import new
from abjad import zaira
import consort
import pitchtools

drum_heartbeat_musicSpecifier = consort.MusicSpecifier(
    attachment_handler = consort.AttachmentHandler(
        dynamic_expression = zaira.materials.midground_dynamic_attachment_expression,
    ),
    pitch_handler = consort.AbsolutePitchHandler(
        pitch_specifier = pitchtools.PitchSegment(
            items = [zaira.makers.Percussion.KICK_DRUM],
        ),
    ),
    rhythm_maker = new(
        zaira.materials.sustained_rhythm_maker,
        inciseSpecifier__talea_denominator = 8,
    ),
)

B.2.8 ZAIRA.MATERIALS.DRUM_STORM_MUSICSpecifier

# -*- encoding: utf-8 -*-
from abjad import indicatortools
from abjad.tools import pitchtools
from abjad.tools import selectortools
from abjad.tools import spannertools
from abjad import new
import consort
import zaira

drum_storm_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.erratic_dynamic_attachment_expression,
        stem_tremolo_spanner=consort.AttachmentExpression(
            attachments=spannertools.StemTremoloSpanner(),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('>\', (1, 16))
        ),
        accent=consort.AttachmentExpression(
            attachments=(
                indicatortools.Articulation('accent'),
            ),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)[0],
        ),
    ),
    pitch_handler=consort.AbsolutePitchHandler(
        pitch_specifier=pitchtools.PitchSegment(
            items={
                zaira.makers.Percussion.HIGH_TOM,
                zaira.makers.Percussion.LOW_TOM,
                zaira.makers.Percussion.BASS_DRUM,
                zaira.makers.Percussion.HIGH_TOM,
                zaira.makers.Percussion.BASS_DRUM,
                zaira.makers.Percussion.LOW_TOM,
                zaira.makers.Percussion.HIGH_TOM,
                zaira.makers.Percussion.BASS_DRUM,
                zaira.makers.Percussion.LOW_TOM,
            },
            rhythm_maker=new(
                zaira.materials.reiterating_rhythm_maker,
                denominators=(16, 16, 4, 16, 4),
                extra_counts_per_division=(0, 1, 0, 1, 2),
            ),
        ),
    ),
)

B.2.9 zaira.materials.drum_tranquilo_musicSpecifier

# -*- encoding: utf-8 -*-
from abjad import new
from abjad.tools import pitchtools
import consort
import zaira
drum_tranquilo_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        laissez_vibrer=zaira.materials.laissez_vibrer_attachment_expression,
    ),
    pitch_handler=consort.AbsolutePitchHandler(
        pitch_specifier=pitchtools.PitchSegment(
            items=(
                zaira.makers.Percussion.HIGH_TOM,
                zaira.makers.Percussion.LOW_TOM,
                zaira.makers.Percussion.BASS_DRUM,
                zaira.makers.Percussion.HIGH_TOM,
                zaira.makers.Percussion.BASS_DRUM,
                zaira.makers.Percussion.LOW_TOM,
                zaira.makers.Percussion.HIGH_TOM,
                zaira.makers.Percussion.BASS_DRUM,
                zaira.makers.Percussion.LOW_TOM,
            ),
        ),
    ),
    rhythm_maker=new(
        zaira.materials.stuttering_rhythm_maker,
        extra_counts_per_division=None,
        incise_specifier__prefix_talea=(1,),
        incise_specifier__prefix_counts=(1,),
        incise_specifier__talea_denominator=16,
    ),
)

B.2.10  ZAIRA.MATERIALS.ERRATIC_DYNAMIC_ATTACHMENT_EXPRESSION

# -*- encoding: utf-8 -*-
from abjad import *
import consort

erratic_dynamic_attachment_expression = consort.AttachmentExpression(
    attachments=(
        consort.SimpleDynamicExpression(
            hairpin_start_token='p',
            hairpin_stop_token='f',
            minimum_duration=durationtools.Duration(1, 8),
        ),
        consort.SimpleDynamicExpression(
            hairpin_start_token='f',
            hairpin_stop_token='p',
            minimum_duration=durationtools.Duration(1, 8),
        ),
        consort.SimpleDynamicExpression(
            hairpin_start_token='mf',
            hairpin_stop_token='o',
            minimum_duration=durationtools.Duration(1, 8),
        ),
    )
B.2.11  

```
from abjad import *

flourish_rhythm_maker = rhythmakertools.TaleaRhythmMaker(
    talea=rhythmakertools.Talea(
        counts=(1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 2, 2,),
        denominator=32,
    ),
    split_divisions_by_counts=None,
    extra_counts_per_division=(0, 0, 0, 1, 0, 0, 1, 1),
    beam_specifier=rhythmakertools.BeamSpecifier(
        beam_each_division=False,
        beam_divisions_together=False,
    ),
    burnish_specifier=rhythmakertools.BurnishSpecifier(
        left_classes=(-1, 1, -1, -1, 1),
        right_classes=(1, -1, 1, -1, 1, -1),
        left_counts=(1,),
        right_counts=(1,),
        outer_divisions_only=True,
    ),
    duration_spelling_specifier=rhythmakertools.DurationSpellingSpecifier(
        decrease_durations_monotonically=True,
    ),
    tie_specifier=rhythmakertools.TieSpecifier(
        tie_across_divisions=True,
    ),
    tie_split_notes=False,
)  
```

B.2.12  

```
from abjad import *
import consort
```
foreground_dynamic_attachment_expression = consort.AttachmentExpression(
    attachments=(
        consort.SimpleDynamicExpression('fff'),
        consort.SimpleDynamicExpression('f'),
        consort.SimpleDynamicExpression('ff'),
        consort.SimpleDynamicExpression('mf'),
    ),
    selector=selectortools.select_pitched_runs()[0],
)

B.2.13 ZAIRA.MATERIALS.GLISSANDO_RHYTHM_MAKER

# -*- encoding: utf-8 -*-
from abjad import *

glissando_rhythm_maker = rhythmmakertools.IncisedRhythmMaker(
    inciseSpecifier=rhythmmakertools.InciseSpecifier(  
        prefix_talea=(1,),
        prefix_counts=(0,),
        suffix_talea=(1,),
        suffix_counts=(1,),
        talea_denominator=16,
        body_ratio=mathtools.Ratio([1]),
        outer_divisions_only=True,
    ),
    beamSpecifier=rhythmmakertools.BeamSpecifier(  
        beam_each_division=False,
        beam_divisions_together=False,
    ),
    durationSpellingSpecifier=rhythmmakertools.DurationSpellingSpecifier(  
        decrease_durations_monotonically=True,
        forbidden_written_duration=durationtools.Duration(1, 2),
    ),
    tupletSpellingSpecifier=rhythmmakertools.TupletSpellingSpecifier(  
        avoid_dots=True,
        is_diminution=True,
        simplify_tuplets=True,
    ),
)

B.2.14 ZAIRA.MATERIALS.GRANULAR_TIMESPAN_MAKER

# -*- encoding: utf-8 -*-
from abjad import *
import consort

granular_timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=rhythmmakertools.Talea(
        counts=(1, 4, 3),
        denominator=16,
    ),
)
playing_talea=rhythmmakertools.Talea(
    counts=(1, 2, 1, 2, 2, 1, 2),
    denominator=16,
),
playing_groupings=(1, 1, 2, 1),
repeat=True,
silence_talea=rhythmmakertools.Talea(
    counts=(4, 8, 7, 9, 2, 13),
    denominator=8,
),
step_anchor=Right,
synchronize_groupings=False,
synchronize_step=False,
timespan_specifier=consort.TimespanSpecifier(
    minimum_duration=None,
),
)

B.2.15  zaira.materials.laissez_vibrer_attachment_expression

laissez_vibrer_attachment_expression = consort.AttachmentExpression(
    attachments=(
        indicatortools.LaissezVibrer(),
        markuptools.Markup(’L.V.’, Up)
        .caps()
        .smaller()
        .parenthesize()
        .override((’padding’, 0.5))
    ),
    selector=selectortools.select_pitched_runs()[-1],
)

B.2.16  zaira.materials.legato_rhythm_maker

legato_rhythm_maker = rhythmmakertools.TaleaRhythmMaker(
    talea=rhythmmakertools.Talea(
        counts=(3, 2, 1, 1, 4, 3, 1, 1, 1, 5, 2, 4),
        denominator=8,
    ),
    # split_divisions_by_counts=(4, 3, 6, 5),
    extra_counts_per_division=(0, 0, 1, 2, 0, 1),
    beam_specifier=rhythmmakertools.BeamSpecifier(
        beam_each_division=False,
    ),
B.2.17  zaira.materials.legato_timespan_maker

```
# -*- encoding: utf-8 -*-
from abjad import *
import consort

legato_timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=rhythmmakertools.Talea(
        counts=(0, 2, 1),
        denominator=8,
    ),
    playing_talea=rhythmmakertools.Talea(
        counts=(4, 5, 4, 3, 7, 6),
        denominator=8,
    ),
    playing_groupings=(3, 4, 2, 2),
    repeat=True,
    silence_talea=rhythmmakertools.Talea(
        counts=(2, 1, 2, 1, 3, 7, 1, 2),
        denominator=8,
    ),
    step_anchor=Right,
    synchronize_groupings=False,
    synchronize_step=False,
    timespanSpecifier=consort.TimespanSpecifier(
        minimum_duration=durationtools.Duration(1, 8),
    ),
)
```

B.2.18  zaira.materials.metal_agitation_musicSpecifier

```
# -*- encoding: utf-8 -*-
from abjad import indicatortools
from abjad import pitchtools
from abjad import selectortools
from abjad.topleveltools import new
import consort
```
import zaira

metal_agitation_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.foreground_dynamic_attachment_expression,
        accent=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.select_first_logical_tie_in_pitched_runs()[0],
        ),
        staccato=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('staccato'),
            selector=selectortools.select_all_but_first_logical_tie_in_pitched_runs()[0],
        ),
    ),
    pitch_handler=consort.AbsolutePitchHandler(
        pitchSpecifier=pitchtools.PitchSegment(
            items=(
                zaira.makers.Percussion.HIGH_CYMBAL,
                zaira.makers.Percussion.LOW_CYMBAL,
                zaira.makers.Percussion.MIDDLE_CYMBAL,
                zaira.makers.Percussion.HIGH_CYMBAL,
                zaira.makers.Percussion.MIDDLE_CYMBAL,
                zaira.makers.Percussion.LOW_CYMBAL,
                zaira.makers.Percussion.HIGH_CYMBAL,
                zaira.makers.Percussion.MIDDLE_CYMBAL,
                zaira.makers.Percussion.TAM_TAM,
                zaira.makers.Percussion.HIGH_CYMBAL,
                zaira.makers.Percussion.LOW_CYMBAL,
                zaira.makers.Percussion.TAM_TAM,
                zaira.makers.Percussion.LOW_CYMBAL,
                zaira.makers.Percussion.MIDDLE_CYMBAL,
                zaira.makers.Percussion.HIGH_CYMBAL,
            ),
        ),
        rhythm_maker=new(
            zaira.materials.stuttering_rhythm_maker,
            extra_counts_per_division=(2, 1, 2, 1, 0, 2, 1, 0),
        ),
    )
)

B.2.19  ZAIRA.MATERIALS.METAL_BUSHED_MUSIC_SPECIFIER

# -*- encoding: utf-8 -*-
from abjad import Markup
from abjad.tools import pitchtools
from abjad.tools import spannertools
from abjad.tools import selectortools
import consort
import zaira

metal_brushed_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
    )
dynamic_expression=zaira.materials.midground_dynamic_attachment_expression,

text_spanner=consort.AttachmentExpression(
    attachments=consort.ComplexTextSpanner(
        markup=Markup(r'\concat { \vstrut brush }')
        .italic()
        .pad_around(0.5)
        .box(),
    ),
    selector=selectortools.Selector().by_leaves(),
),
stem_tremolo_spanner=consort.AttachmentExpression(
    attachments=(
        spannertools.StemTremoloSpanner(),
        None,
    ),
    selector=selectortools.select_pitched_runs(),
),

pitch_handler=consort.AbsolutePitchHandler(
    pitch_specifier=pitchtools.PitchSegment(
        items=(
            zaira.makers.Percussion.HIGH_CYMBAL,
            zaira.makers.Percussion.LOW_CYMBAL,
            zaira.makers.Percussion.MIDDLE_CYMBAL,
            zaira.makers.Percussion.HIGH_CYMBAL,
            zaira.makers.Percussion.MIDDLE_CYMBAL,
            zaira.makers.Percussion.LOW_CYMBAL,
            zaira.makers.Percussion.HIGH_CYMBAL,
            zaira.makers.Percussion.TAM_TAM,
            zaira.makers.Percussion.MIDDLE_CYMBAL,
            zaira.makers.Percussion.LOW_CYMBAL,
            zaira.makers.Percussion.HIGH_CYMBAL,
            zaira.makers.Percussion.TAM_TAM,
            zaira.makers.Percussion.LOW_CYMBAL,
            zaira.makers.Percussion.MIDDLE_CYMBAL,
            ),
        ),
    ),
    rhythm_maker=zaira.materials.sustained_rhythm_maker,
)

B.2.20  ZAIRA.MATERIALS.METAL_TRANQUILO_MUSICSpecifier

# -*- encoding: utf-8 -*-
from abjad import new
from abjad.tools import pitchtools
import consort
import zaira

metal_tranquilo_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        laissez_vibrer=zaira.materials.laissez_vibrer_attachment_expression,
12 },
13     pitch_handler=consort.AbsolutePitchHandler(
14         pitchSpecifier=pitchtools.PitchSegment(
15             items=(
16                 zaira.makers.Percussion.HIGH_CYMBAL,
17                 zaira.makers.Percussion.LOW_CYMBAL,
18                 zaira.makers.Percussion.MIDDLE_CYMBAL,
19                 zaira.makers.Percussion.HIGH_CYMBAL,
20                 zaira.makers.Percussion.MIDDLE_CYMBAL,
21                 zaira.makers.Percussion.LOW_CYMBAL,
22                 zaira.makers.Percussion.HIGH_CYMBAL,
23                 zaira.makers.Percussion.TAM_TAM,
24                 zaira.makers.Percussion.MIDDLE_CYMBAL,
25                 zaira.makers.Percussion.LOW_CYMBAL,
26                 zaira.makers.Percussion.HIGH_CYMBAL,
27                 zaira.makers.Percussion.TAM_TAM,
28                 zaira.makers.Percussion.LOW_CYMBAL,
29                 zaira.makers.Percussion.MIDDLE_CYMBAL,
30             ),
31         ),
32     ),
33     rhythm_maker=new(
34         zaira.materials.stuttering_rhythm_maker,
35         extra_counts_per_division=None,
36         incise_specifier__prefix_talea=(1,),
37         incise_specifier__prefix_counts=(1,),
38         incise_specifier__talea_denominator=16,
39     ),
40 )

B.2.21 Zaira.Materials.Midground_Dynamic_Attachment_Expression

# -*- encoding: utf-8 -*-
from abjad import *
import consort

midground_dynamic_attachment_expression = consort.AttachmentExpression(
    attachments=(
        consort.SimpleDynamicExpression('mf'),
        consort.SimpleDynamicExpression('mp'),
    ),
    selector=selectortools.select_pitched_runs()[0],
)

B.2.22 Zaira.Materials.Oboe_Solo_MusicSpecifier

# -*- encoding: utf-8 -*-
from abjad.tools import indicatortools
from abjad.tools import spannertools
from abjad.tools import selectortools
from abjad.tools.topleveltools import new
import consort
import zaira
```python
# B.2.23 ZAIRA.MATERIALS.OVERPRESSURE_ATTACHMENT_EXPRESSION

# -*- encoding: utf-8 -*-
from abjad.tools import selectortools
import consort

overpressure_attachment_expression = consort.AttachmentExpression(
    attachments=consort.ComplexTextSpanner(
        markup=r'\filled-box \#(-0.5 . 1.5) \#(-1 . 1) \#0.25',
    ),
    selector=selectortools.Selector(),
)
```

```
oboë_solo_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.erratic_dynamic_attachment_expression,
        trill_spanner=consort.AttachmentExpression(
            attachments=(
                spannertools.ComplexTrillSpanner(interval='+m3'),
                spannertools.ComplexTrillSpanner(interval='+m3'),
                spannertools.ComplexTrillSpanner(interval='+M2'),
            ),
            selector=selectortools.Selector()
        ).by_logical_tie(pitched=True)
        ).by_duration('>', (1, 32))
    ),
    staccato=consort.AttachmentExpression(
        attachments=indicatortools.Articulation('staccato'),
        selector=selectortools.Selector()
    ),
    accent=consort.AttachmentExpression(
        attachments=(
            indicatortools.Articulation('accent'),
            ),
        selector=selectortools.Selector()
    ),
    pitch_handler=consort.AbsolutePitchHandler(
        pitch_specifier="d''' d''' ef''' d''' ef''' f'''' d'''' g'''' d'''' d'''' as'''',
        pitch_application_rate='division',
    ),
    rhythm_maker=new(
        zaira.materials.reiterating_rhythm_maker,
        denominators=(32, 1, 32, 32, 1),
    )
)
B.2.24  zaira.materials.pedals_timespan_maker

```python
# -*- encoding: utf-8 -*-
from abjad import *
import consort

pedals_timespan_maker = consort.DependentTimespanMaker(
    labels=('pedaled',),
    voice_names=('Piano Upper Voice', 'Piano Lower Voice'),
)
```

B.2.25  zaira.materials.percussion_brushed_musicSpecifier

```python
# -*- encoding: utf-8 -*-
from abjad import Markup
from abjad.tools import selectortools
import consort
import zaira

percussion_brushed_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        text_spanner=consort.AttachmentExpression(
            attachments=consort.ComplexTextSpanner(
                markup=Markup(r'\concat { \vstrut brush }')
                                   .italic()
                                   .pad_around(0.5)
                                   .box(),
            ),
            selector=selectortools.Selector().by_leaves(),
        ),
    ),
    pitch_handler=consort.AbsolutePitchHandler(),
    rhythm_maker=zaira.materials.sustained_rhythm_maker,
)
```

B.2.26  zaira.materials.percussion_brushed_tremolo_musicSpecifier

```python
# -*- encoding: utf-8 -*-
from abjad import Markup
from abjad import spannertools
from abjad import selectortools
import consort
import zaira

percussion_brushed_tremolo_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        text_spanner=consort.AttachmentExpression(
            attachments=consort.ComplexTextSpanner(
                markup=Markup(r'\concat { \vstrut brush }')
                                   .italic()
                                   .pad_around(0.5)
                                   .box(),
            ),
            selector=selectortools.Selector().by_leaves(),
        ),
    ),
    pitch_handler=consort.AbsolutePitchHandler(),
    rhythm_maker=zaira.materials.sustained_rhythm_maker,
)
```
percussion_brushed_tremolo_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.erratic_dynamic_attachment_expression,
        text_spanner=consort.AttachmentExpression(
            attachments=consort.ComplexTextSpanner(
                markup=Markup(r'\concat \{ \vstrut brush \}')
                .italic()
                .pad_around(0.5)
                .box(),
            ),
            selector=selectortools.Selector().by_leaves(),
        ),
        stem_tremolo_spanner=consort.AttachmentExpression(
            attachments=spannertools.StemTremoloSpanner(),
            selector=selectortools.Selector().by_leaves(),
        ),
    ),
    pitch_handler=consort.AbsolutePitchHandler(),
    rhythm_maker=zaira.materials.sustained_rhythm_maker,
)

B.2.27  ZAIRA.MATERIALS PERCUSSION_FANFARE_MUSIC_SPECIFIER

percussion_fanfare_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=consort.AttachmentExpression(
            attachments=indicatortools.Dynamic('fff'),
            selector=selectortools.select_pitched_runs()[0],
        ),
        accent=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector()
            .by_leaves()
            .by_logical_tie(pitched=True,
            )[0],
        ),
    ),
    pitch_handler=consort.AbsolutePitchHandler(),
    rhythm_maker=zaira.materials.reiterating_rhythm_maker,
)

B.2.28  ZAIRA.MATERIALS PERCUSSION_REITERATION_MUSIC_SPECIFIER
import consort
import zaira

percussion_reiteration_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=consort.AttachmentExpression(
            attachments=indicatortools.Dynamic('ppp'),
            selector=selectortools.select_pitched_runs()[:] ,
        ),
        staccato=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('.'),
            selector=selectortools.Selector()
                .by_leaves(
                    ).by_logical_tie(pitched=True
                        )(0],
                ),
        ),
    ),
    pitch_handler=consort.AbsolutePitchHandler(),
    rhythm_maker=zaira.materials.reiterating_rhythm_maker,
)

B.2.29  Zaira.materials.percussion_superball_musicSpecifier

# -*- encoding: utf-8 -*-
from abjad import Markup
from abjad.tools import selectortools
import consort
import zaira

percussion_superball_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        text_spanner=consort.AttachmentExpression(
            attachments=consort.ComplexTextSpanner(
                markup=Markup(r'\concat { \vstrut superball }')
                    .italic()
                        .pad_around(0.5)
                            .box(),
                    ),
                selector=selectortools.Selector().by_leaves(),
                    ),
    ),
    pitch_handler=consort.AbsolutePitchHandler(),
    rhythm_maker=zaira.materials.sustained_rhythm_maker,
)

B.2.30  Zaira.materials.piano_clusters_musicSpecifier

# -*- encoding: utf-8 -*-
from abjad.tools import pitchtools
import consort
import zaira

piano_clusters_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
    ),
    labels='pedaled',
    pitch_handler=consort.PitchClassPitchHandler(
        logical_tie_expressions=(
            consort.KeyClusterExpression(
                staff_space_width=7,
            ),
            consort.KeyClusterExpression(
                staff_space_width=7,
            ),
            consort.KeyClusterExpression(
                staff_space_width=9,
            ),
            consort.KeyClusterExpression(
                include_black_keys=False,
                staff_space_width=7,
            ),
            consort.KeyClusterExpression(
                staff_space_width=7,
            ),
            consort.KeyClusterExpression(
                include_white_keys=False,
                staff_space_width=9,
            ),
            consort.KeyClusterExpression(
                staff_space_width=9,
            ),
            consort.KeyClusterExpression(
                staff_space_width=9,
            ),
            consort.KeyClusterExpression(
                staff_space_width=7,
            ),
        ),
        pitch_specifier="c g e a",
        pitch_range=pitchtools.PitchRange('[A1, C7]'),
        register_specifier=zaira.materials.register_specifier_inventory[0],
        rhythm_maker=zaira.materials.stuttering_rhythm_maker,
    )
)

B.2.31  ZAIRA.MATERIALS.PIANO_DRONE_MUSIC_SPECIFIER

# -*- encoding: utf-8 -*-
from abjad.tools import spannertools
from abjad.tools import selectortools
import consort
import zaira

piano_drone_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        stem_tremolo_spanner=consort.AttachmentExpression(
            )
        )
)
B.2.32  ZAIRA.MATERIALS.PIANO_FANFARE_MUSIC_SPECIFIER

from abjad.tools import indicatortools
from abjad.tools import selectortools
from abjad.tools import pitchtools
import consort
import zaira

piano_fanfare_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=consort.AttachmentExpression(
            attachments=indicatortools.Dynamic('fff'),
            selector=selectortools.select_pitched_runs()[0],
        ),
        accent=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector().by_leaves().by_logical_tie(pitched=True,
            )[0],
        ),
    ),
    labels='pedaled',
    pitch_handler=consort.PitchClassPitchHandler(
        logical_tie_expressions=(
            consort.KeyClusterExpression(
                include_black_keys=False,
                staff_space_width=7,
            ),
            consort.KeyClusterExpression(
                staff_space_width=9,
            ),
            consort.KeyClusterExpression(
                staff_space_width=7,
            ),
        )
    )}
B.2.33  ZAIRA.MATERIALS.PIANO_FLOURISH_MUSICSpecifier

```python
# -*- encoding: utf-8 -*-
from abjad.tools import pitchtools
from abjad.tools import spannertools
from abjad.tools import selectortools
import consort
import zaira

piano_flourish_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.erratic_dynamic_attachment_expression,
        slur=consort.AttachmentExpression(
            attachments=spannertools.Slur(),
            selector=selectortools.select_pitched_runs(),
        ),
        labels='pedaled',
    ),
    pitchSpecifier="c a f d e b",
    pitchRange=pitchtools.PitchRange('[A1, C7)'),
    registerSpecifier=consort.RegisterSpecifier(),
)
```

rhythm_maker=zaira.materials.reiterating_rhythm_maker,
pitch_handler=consort.PitchClassPitchHandler(
    pitch_specifier='d f e g cs as',
    pitch_range=pitchtools.PitchRange('[A1, C7)'),
    registerSpecifier=zaira.materials.register_specifier_inventory[2],
    rhythm_maker=zaira.materials.flourish_rhythm_maker,
)

B.2.34  Zaira.materials.piano_guero_musicSpecifier

slow_markup = Markup(r'\concat { \vstrut slow }', Up)
slow_markup = slow_markup.italic()
slow_markup = slow_markup.pad_around(0.5)
slow_markup = slow_markup.box()

fast_markup = Markup(r'\concat { \vstrut fast }', Up)
fast_markup = fast_markup.italic()
fast_markup = fast_markup.pad_around(0.5)
fast_markup = fast_markup.box()

piano_guero_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        clef_spanner=consort.ClefSpanner('percussion'),
        dynamic_expression=zaira.materials.midground_dynamic_attachment_expression,
        staff_lines_spanner=spannertools.StaffLinesSpanner(
            lines=(4, -4),
            overrides={
                'note_head__no_ledgers': True,
                'note_head__style': 'cross',
            }
        ),
    text_spanner=consort.AttachmentExpression(
        attachments=consort.ComplexTextSpanner(
            markup=Markup(r'\concat { \vstrut guero }')
            .italic()
            .pad_around(0.5)
            .box(),
            selector=selectortools.Selector().by_leaves(),
        ),
        direction_markup=consort.AttachmentExpression(
            attachments=(
                slow_markup,
                slow_markup,
                fast_markup,
                slow_markup,
            ))
    )
)
B.2.35  
Zaira.materials.piano_pedals_musicSpecifier

```python
# -*- encoding: utf-8 -*-
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
import consort

piano_pedals_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        piano_pedal_spanner=consort.AttachmentExpression(
            attachments=(
                consort.ComplexPianoPedalSpanner(),
            ),
            selector=selectortools.Selector(),
        ),
    ),
    rhythm_maker=rhythmmakertools.SkipRhythmMaker(
        duration_spellingSpecifier=rhythmmakertools.DurationSpellingSpecifier(
            forbid_meter_rewrite=True,
        ),
    ),
)
```

B.2.36  
Zaira.materials.piano_prepared_bass_musicSpecifier

```python
# -*- encoding: utf-8 -*-
from abjad import Markup
from abjad.tools import indicatortools
from abjad.tools import pitchtools
from abjad.tools import selectortools
from abjad.tools import spannertools
import consort
import zaira

staccato_selector = selectortools.Selector()
staccato_selector = staccato_selector.by_leaves()
staccato_selector = staccato_selector.by_logical_tie(pitched=True,
)
staccato_selector = staccato_selector.by_duration('<=', (1, 16))
sustain_selector = selectortools.Selector()
sustain_selector = sustain_selector.by_leaves()
```
```
sustain_selector = sustain_selector.by_logical_tie(
    pitched=True,
)
sustain_selector = sustain_selector.by_duration('>', (1, 16))

piano_prepared_bass_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        clef_spanner=consort.AttachmentExpression(
            clef='bass_15',
        ),
        selector=selectortools.Selector(),
    ),
    text_spanner=consort.AttachmentExpression(
        attachments=consort.ComplexTextSpanner(
            markup=Markup(r'\concat { \vstrut prepared }')
                .italic()
                .pad_around(0.5)
                .box(),
            overrides={
                'note_head__style': 'cross',
            },
            selector=selectortools.Selector(),
        ),
        staccato=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('.'),
            selector=staccato_selector,
        ),
        trill_spanner=consort.AttachmentExpression(
            attachments=spannertools.ComplexTrillSpanner(
                interval='m2',
            ),
            selector=sustain_selector,
        ),
    ),
    labels='pedaled',
    pitch_handler=consort.AbsolutePitchHandler(
        pitchSpecifier=pitchtools.PitchSegment(
            'A0 C1 B0 D1 C#1 E1 D#1 F1 G1 A#0 F#1',
        ),
    ),
    rhythm_maker=zaira.materials.undergrowth_rhythm_maker,
)

B.2.37  ZAIRA.MATERIALS.PIANO_PREPARED_TREBLE_MUSIC_SPECIFIER
```
from abjad.tools import spannertools
import consort
import zaira

staccato_selector = selectortools.Selector()
staccato_selector = staccato_selector.by_leaves()
staccato_selector = staccato_selector.by_logical_tie(
    pitched=True,
)
staccato_selector = staccato_selector.by_duration('<=', (1, 16))

sustain_selector = selectortools.Selector()
sustain_selector = sustain_selector.by_leaves()
sustain_selector = sustain_selector.by_logical_tie(
    pitched=True,
)
sustain_selector = sustain_selector.by_duration('>', (1, 16))

piano_prepared_treble_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        clef_spanner=consort.AttachmentExpression(
            attachments=consort.ClefSpanner(
                clef='treble^15',
                ),
            selector=selectortools.Selector(),
        ),
        text_spanner=consort.AttachmentExpression(
            attachments=consort.ComplexTextSpanner(
                markup=Markup(r'\concat { \vstrut prepared }')
                .italic()
                .pad_around(0.5)
                .box(),
                overrides={
                    'note_head__style': 'cross',
                },
            ),
            selector=selectortools.Selector(),
        ),
        staccato=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('.'),
            selector=staccato_selector,
        ),
        trill_spanner=consort.AttachmentExpression(
            attachments=spannertools.ComplexTrillSpanner(
                interval='m2',
            ),
            selector=sustain_selector,
        ),
    labels='pedaled',
)
pitch_handler=consort.AbsolutePitchHandler(
   pitchSpecifier=pitchtools.PitchSegment(
      'C7 B7 G#7 E7 F7 G7 A#7 F#7 A7',
   ),
   ),
rhythm_maker=zaira.materials.undergrowth_rhythm_maker,
)

B.2.38  zaira.materials.proportions

# -*- encoding: utf-8 -*-
from abjad import *
import consort

proportions = consort.Proportions(
    [
        [2],  # A
        [1, 6, 3],  # B
        [7],  # C
        [1, 2, 1],  # D
        [1, 8, 4],  # E
        [1, 14, 7],  # F
        [1, 4, 2],  # G
        [5],  # H
        [1, 15],  # I
        [2],  # J
        [3],  # K
    ]
)

B.2.39  zaira.materials.registerSpecifier_inventory

# -*- encoding: utf-8 -*-
from abjad.tools import mathtools
import consort

registerSpecifier_inventory = consort.RegisterSpecifierInventory(
    [
        consort.RegisterSpecifier(
            divisionInfections=consort.RegisterInfectionInventory(
                consort.RegisterInfection(
                    inflections=(-3, 3),
                    ratio=mathtools.Ratio([1]),
                ),
                consort.RegisterInfection(
                    inflections=(3, 6, -3),
                    ratio=mathtools.Ratio([2, 1]),
                ),
                consort.RegisterInfection(
                    inflections=(-6, 0, 6),
                    ratio=mathtools.Ratio([1, 2]),
                ),
        ),
    ]
)
phrase_inflections=consort.RegisterInflectionInventory([  
consort.RegisterInflection(  
    inflections=(6, 0, -6),  
    ratio=mathtools.Ratio([2, 1]),  
  ),  
  consort.RegisterInflection(  
    inflections=(3, 0, -3),  
    ratio=mathtools.Ratio([1, 2]),  
  ),  
],),  

segment_inflections=consort.RegisterInflectionInventory([  
consort.RegisterInflection(  
    inflections=(-12, 0, 12),  
    ratio=mathtools.Ratio([2, 1]),  
  ),  
],),  

consort.RegisterSpecifier(  
  division_inflections=consort.RegisterInflectionInventory([  
    consort.RegisterInflection(  
      inflections=(-6, 6),  
      ratio=mathtools.Ratio([1]),  
    ),  
    consort.RegisterInflection(  
      inflections=(-12, 3, 9),  
      ratio=mathtools.Ratio([2, 1]),  
    ),  
    consort.RegisterInflection(  
      inflections=(6, -6),  
      ratio=mathtools.Ratio([1]),  
    ),  
  ]),  
  phrase_inflections=consort.RegisterInflectionInventory([  
    consort.RegisterInflection(  
      inflections=(3, -3),  
      ratio=mathtools.Ratio([1]),  
    ),  
    consort.RegisterInflection(  
      inflections=(-3, 3),  
      ratio=mathtools.Ratio([1]),  
    ),  
  ]),  
  segment_inflections=consort.RegisterInflectionInventory([  
    consort.RegisterInflection(  
      inflections=(12, 6, 0, -12),  
      ratio=mathtools.Ratio([3, 2, 1]),  
    ),  
],),  
})
76       ratio=mathtools.Ratio([1]),
77   ),
78   consort.RegisterInflection(
79       inflections=(3, -3),
80       ratio=mathtools.Ratio([1]),
81   ),
82   consort.RegisterInflection(
83       inflections=(-3, 3),
84       ratio=mathtools.Ratio([1]),
85   ),
86 ),
87   phrase_inflections=consort.RegisterInflectionInventory([
88   consort.RegisterInflection(
89       inflections=(6, -6),
90       ratio=mathtools.Ratio([1]),
91   ),
92   consort.RegisterInflection(
93       inflections=(-3, 3, 6),
94       ratio=mathtools.Ratio([2, 1]),
95   ),
96 ]),
97   segment_inflections=consort.RegisterInflectionInventory([
98   consort.RegisterInflection(
99       inflections=(-6, 6, -3),
100      ratio=mathtools.Ratio([2, 1]),
101   ),
102   ],
103   ),
104   consort.RegisterSpecifier(
105        division_inflections=consort.RegisterInflectionInventory([
106           consort.RegisterInflection(
107               inflections=(-3, 3),
108               ratio=mathtools.Ratio([1]),
109           ),
110           consort.RegisterInflection(
111               inflections=(-3, 3),
112               ratio=mathtools.Ratio([1]),
113           ),
114           consort.RegisterInflection(
115               inflections=(3, -3),
116               ratio=mathtools.Ratio([1]),
117           ),
118         ]),
119        phrase_inflections=consort.RegisterInflectionInventory([
120           consort.RegisterInflection(
121               inflections=(-3, 3),
122               ratio=mathtools.Ratio([1]),
123           ),
124           consort.RegisterInflection(
125               inflections=(6, 3, -6),
126               ratio=mathtools.Ratio([2, 1]),
127           ),
128         ]),
129        segment_inflections=consort.RegisterInflectionInventory([
consort.RegisterInflection(
    inflections=(-12, 6, -6, 12),
    ratio=mathtools.Ratio([2, 1, 3]),
)
}
)
)
}

B.2.40  ZAIRA.MATERIALS.REITERATING_RHYTHM MAKER

# -*- encoding: utf-8 -*-
from abjad import *

reiterating_rhythm_maker = rhythmtools.EachDivisionRhythmMaker(
    denominators=(16,),
    beam_specifier=rhythmtoolsBeamSpecifier(
        beam_each_division=False,
        beam_divisions_together=False,
    ),
    duration_spelling_specifier=rhythmtools.DurationSpellingSpecifier(
        decrease_durations_monotonically=True,
        forbidden_written_duration=durationtools.Duration(1, 2),
    ),
)

B.2.41  ZAIRA.MATERIALS.SPARSE_TIMESSPAN MAKER

# -*- encoding: utf-8 -*-
from abjad import *

import consort

sparse_timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=rhythmtools.Talea(
        counts=(0, 4, 2, 6, 3),
        denominator=16,
    ),
    playing_talea=rhythmtools.Talea(
        counts=(4, 6, 8, 5, 6, 6, 4),
        denominator=16,
    ),
    playing_groupings=(1, 1, 2, 1, 2, 1, 1, 1),
    repeat=True,
    silence_talea=rhythmtools.Talea(
        counts=(20, 12, 26, 10, 14, 7),
        denominator=16,
    ),
    step_anchor=Right,
    synchronize_groupings=False,
    synchronize_step=False,
    timespan_specifier=consort.TimespanSpecifier(
        minimum_duration=durationtools.Duration(1, 8),
    ),
B.2.42  zaira.materials.string_chord_musicSpecifier

```python
1 # -*- encoding: utf-8 -*-
2 import consort
3 import zaira
4
5 string_chord_musicSpecifier = consort.MusicSpecifier(
6     attachment_handler=consort.AttachmentHandler(
7         dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
8     ),
9     pitch_handler=consort.PitchClassPitchHandler(
10        pitch_specifier='c ef d',
11        register_specifier=consort.RegisterSpecifier(),
12     ),
13     rhythm_maker=zaira.materials.sustained_rhythm_maker,
14 )
```

B.2.43  zaira.materials.string_flourish_musicSpecifier

```python
1 # -*- encoding: utf-8 -*-
2 from abjad import Markup
3 from abjad.tools import selectortools
4 from abjad.tools import spannertools
5 from abjad.tools.topleveltools import new
6 import consort
7 import zaira
8
9 string_flourish_musicSpecifier = consort.MusicSpecifier(
10     attachment_handler=consort.AttachmentHandler(
11         dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
12         slur=consort.AttachmentExpression(
13             attachments=spannertools.Slur(),
14         ),
15         selector=selectortools.select_pitched_runs(),
16     ),
17     text_spanner=consort.AttachmentExpression(
18         attachments=consort.ComplexTextSpanner(
19             markup=Markup(r'\concat { \vstrut flautando }')
20             .italic()
21             .pad_around(0.5)
22             .box(),
23         ),
24         selector=selectortools.select_pitched_runs(),
25     ),
26 ),
27     pitch_handler=consort.PitchClassPitchHandler(
28        pitch_specifier='d f e g cs as',
29        register_specifier=zaira.materials.register_specifier_inventory[2],
30        register_spread=0,
31 )
```
rhythm_maker=new(
    zaira.materials.flourish_rhythm_maker,
    tieSpecifier__tie_across_divisions=False,
),
)

B.2.44  

```python
# -*- encoding: utf-8 -*-
from abjad.tools import markuptools
from abjad.tools import schemetools
from abjad.tools import selectortools
from abjad.tools import spannertools
import consort
import zaira

fourth_spanner = spannertools.ComplexTrillSpanner(
    interval='+P4',
    overrides={
        'trill_pitch_head__stencil': schemetools.Scheme(
            'ly:text-interface::print',
        ),
        'trill_pitch_head__text': markuptools.Markup.musicglyph(
            'noteheads.s0harmonic',
            direction=None,
        ),
    },
)

third_spanner = spannertools.ComplexTrillSpanner(
    interval='m3',
    overrides={
        'trill_pitch_head__stencil': schemetools.Scheme(
            'ly:text-interface::print',
        ),
        'trill_pitch_head__text': markuptools.Markup.musicglyph(
            'noteheads.s0harmonic',
            direction=None,
        ),
    },
)

string_trills_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        trill_spanner=consort.AttachmentExpression(
            attachments=(
                third_spanner,
                fourth_spanner,
            ),
            selector=selectortools.select_pitched_runs(),
        ),
    ),
)
```python
pitch_handler=consort.PitchClassPitchHandler(
    pitch_specifier='a c b',
    register_specifier=zaira.materials.register_specifier_inventory[3],
),

rhythm_maker=zaira.materials.sustained_rhythm_maker,
)
```

**B.2.45** zaira.materials.string_tutti_overpressure_musicSpecifier

```python
from abjad import Markup
from abjad.tools import indicatortools
from abjad.tools import selectortools
import consort
import zaira

string_tutti_overpressure_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        text_spanner=consort.AttachmentExpression(
            attachments=consort.ComplexTextSpanner(
                markup=Markup(r'\concat { \vstrut overpressure }')
                .italic()
                .pad_around(0.5)
                .box(),
            ),
            selector=selectortools.Selector().by_leaves(),
        ),
        dynamic_and_accent=consort.AttachmentExpression(
            attachments=(
                indicatortools.Dynamic('fff'),
                indicatortools.Articulation('accent'),
                indicatortools.Articulation('tenuto'),
            ),
            selector=selectortools.Selector()
                .by_leaves()
                .by_logical_tie(pitched=True)[0],
        ),
    ),
    pitch_handler=consort.AbsolutePitchHandler(
        logical_tie_expressions=(
            consort.ChordExpression(
                chord_expr=(0, 7),
            ),
        ),
        rhythm_maker=zaira.materials.sustained_rhythm_maker,
    )
```
from abjad import Markup
from abjad.tools import indicatortools
from abjad.tools import selectortools
from abjad.tools import spannertools
import consort
import zaira

string_undergrowth_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.midground_dynamic_attachment_expression,
        text_spanner=consort.AttachmentExpression(
            attachments=consort.ComplexTextSpanner(
                markup=Markup(r'\concat { \vstrut overpressure }')
                .italic()
                .pad_around(0.5)
                .box(),
                ),
            selector=selectortools.Selector().by_leaves(),
            ),
        accent_and_tenuto=consort.AttachmentExpression(
            attachments=(
                indicatortools.Articulation('accent'),
                indicatortools.Articulation('tenuto'),
            ),
            selector=selectortools.Selector()
            .by_leaves()
            .by_logical_tie(pitched=True)[0],
            ),
        stem_tremolo_spanner=consort.AttachmentExpression(
            attachments=(
                None,
                None,
                spannertools.StemTremoloSpanner(),
            ),
            selector=selectortools.Selector()
            .by_leaves()
            .by_logical_tie(pitched=True)
            ),
        ),
    pitch_handler=consort.PitchClassPitchHandler(
        pitch_specifier='a c b d',
        register_specifier=zaira.materials.register_specifier_inventory[3],
        ),
    rhythm_maker=zaira.materials.undergrowth_rhythm_maker,
)
B.2.47  
```python
# -*- encoding: utf-8 -*-
from abjad import *

stuttering_rhythm_maker = rhythmmakertools.IncisedRhythmMaker(
    inciseSpecifier=rhythmmakertools.InciseSpecifier(
        outer_divisions_only=False,
        prefix_talea=(1, 1, 1, 2, 2, 1, 2),
        prefix_counts=(2, 2, 1, 2, 3, 2, 2, 2, 1),
        suffix_talea=(1,),
        suffix_counts=(0,),
        talea_denominator=16,
        fill_with_notes=False,
    ),
    extra_counts_per_division=(0, 0, 1, 0, 1),
    beamSpecifier=rhythmmakertools.BeamSpecifier(
        beam_each_division=False,
        beam_divisions_together=False,
    ),
    tuplet_spelling_specifier=rhythmmakertools.TupletSpellingSpecifier(
        avoid_dots=True,
        is_diminution=True,
        simplify_tuplets=True,
    ),
)
```

B.2.48  
```python
# -*- encoding: utf-8 -*-
from abjad import *

sustained_rhythm_maker = rhythmmakertools.NoteRhythmMaker(
    beamSpecifier=rhythmmakertools.BeamSpecifier(
        beam_each_division=False,
        beam_divisions_together=False,
    ),
    tieSpecifier=rhythmmakertools.TieSpecifier(
        tie_across_divisions=(True, True, False),
    ),
)
```

B.2.49  
```python
# -*- encoding: utf-8 -*-
from abjad import *
import consort

sustained_timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=rhythmmakertools.Talea(
        counts=(0, 2, 1),
    ),
)```
denominator=8,
),
playing_talea=rhythmmakertools.Talea(
    counts=(4, 5, 4, 3, 7, 6),
    denominator=8,
),
playing_groupings=(3, 4, 2, 2, 3),
repeat=True,
silence_talea=rhythmmakertools.Talea(
    counts=(2, 1, 2, 1, 3, 7, 1, 2),
    denominator=8,
),
step_anchor=Right,
synchronize_groupings=False,
synchronize_step=False,
timespan_specifier=consort.TimespanSpecifier(
    minimum_duration=durationtools.Duration(1, 8),
),
)

B.2.50  ZAIRA.MATERIALS.TEMPI

from abjad import *

tempi = indicatortools.TempoInventory(
    [
        indicatortools.Tempo(
            duration=durationtools.Duration(1, 4),
            units_per_minute=72,
        ),
        indicatortools.Tempo(
            duration=durationtools.Duration(1, 4),
            units_per_minute=48,
        ),
        indicatortools.Tempo(
            duration=durationtools.Duration(1, 4),
            units_per_minute=108,
        ),
    ]
)

B.2.51  ZAIRA.MATERIALS.TIME_SIGNATURES

from abjad.tools import indicatortools

time_signatures = indicatortools.TimeSignatureInventory(
    [
        (2, 4),
        (3, 4),
        (3, 8),
    ]
)
B.2.52  zaira.materials.total_duration_in_seconds

```python
# -*- encoding: utf-8 -*-
from abjad import *

total_duration_in_seconds = 480
```

B.2.53  zaira.materials.tutti_timespan_maker

```python
# -*- encoding: utf-8 -*-
from abjad import *
import consort

tutti_timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=None,
    playing_talea=rhythmmakertools.Talea(
        counts=(4, 6, 6, 5),
        denominator=16,
    ),
    playing_groupings=(1, 2, 1, 3, 2, 2),
    repeat=True,
    silence_talea=rhythmmakertools.Talea(
        counts=(8, 6, 10, 7, 12),
        denominator=8,
    ),
    step_anchor=Right,
    synchronize_groupings=True,
    synchronize_step=True,
    timespan_specifier=consort.TimespanSpecifier(
        minimum_duration=durationtools.Duration(1, 8),
    ),
)
```

B.2.54  zaira.materials.undergrowth_rhythm_maker

```python
# -*- encoding: utf-8 -*-
from abjad import *

undergrowth_rhythm_maker = rhythmmakertools.TaleaRhythmMaker(
    talea=rhythmmakertools.Talea(
        counts=(1, -3, 1, -2, 1, -2, 2, -4, 4),
        denominator=16,
    ),
    extra_counts_per_division=(0, 2, 1, 0, 1, 1, 0),
)```
B.2.55  ZIRA.MATERIALS.WIND_AIRTONE_MUSICSpecifier

```python
# -*- encoding: utf-8 -*-
from abjad import Markup
from abjad.tools import selectortools
import consort
import zaira

wind_airtone_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        text_spanner=consort.AttachmentExpression(
            attachments=consort.ComplexTextSpanner(
                markup=Markup(r'\concat { \vstrut airtone }')
            .italic()
            .pad_around(0.5)
            .box(),
            overrides={
                'note_head__style': 'slash',
            }
        ),
    ),
    selector=selectortools.Selector().by_leaves(),
),
    pitch_handler=consort.PitchClassPitchHandler(
        pitchSpecifier='a c b d b f gs e',
        registerSpecifier=consort.RegisterSpecifier(),
        registerSpread=0,
    ),
    rhythmMaker=zaira.materials.sustained_rhythmMaker,
)
```

B.2.56  ZIRA.MATERIALS.WIND_KEYCLICK_MUSICSpecifier

```python
# -*- encoding: utf-8 -*-
from abjad import Markup
from abjad.tools import indicatorTools
from abjad.tools import spannertools
from abjad.tools import selectortools
import consort
import zaira
```

```python
```
wind_keyclick_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        clef_spanner=consort.ClefSpanner('percussion'),
        staff_lines_spanner=spannertools.StaffLinesSpanner(
            lines=(4, -4),
            overrides={
                'note_head__no_ledgers': True,
                'note_head__style': 'cross',
            },
        ),
    ),
    text_spanner=consort.AttachmentExpression(
        attachments=consort.ComplexTextSpanner(
            markup=Markup(r'\concat \vstrut \keyclick')
                .italic()
                .pad_around(0.5)
                .box(),
        ),
        selector=selectortools.Selector().by_leaves(),
    ),
    stem_tremolo_spanner=consort.AttachmentExpression(
        attachments=spannertools.StemTremoloSpanner(),
        selector=selectortools.Selector(
            ).by_logical_tie(pitched=True
                ).by_duration('>', (1, 16))
        ),
    ),
    staccato=consort.AttachmentExpression(
        attachments=indictortools.Articulation('.'),
        selector=selectortools.Selector(
            ).by_logical_tie(pitched=True
                ).by_duration('<', (1, 8)
                    ).by_length(1)
        ),
    ),
    pitch_handler=consort.AbsolutePitchHandler(
        pitch_specifier="c' g' f g' g' c' f c' f g' c' f g'",
        pitches_are_nonsemantic=True,
    ),
    rhythm_maker=zaira.materials.undergrowth_rhythm_maker,
)
```
wind_slap_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.midground_dynamic_attachment_expression,
        override_spanner=consort.AttachmentExpression(
            attachments=spannertools.Spanner(
                overrides={
                    'note_head__stencil': schemetools.Scheme(
                        ly:shared:print
                    ),
                    'note_head__text': markuptools.Markup.triangle(False)
                        .rotate(180)
                        .scale((0.667, 0.667))
                        .override({'thickness': 0.5})
                        .translate((0, -0.9)),
                ),
            selector=selectortools.select_pitched_runs(),
        ),
    text_spanner=consort.AttachmentExpression(
        attachments=consort.ComplexTextSpanner(
            markup=Markup(r'\concat { \vstrut slap }')
                .italic()
                .pad_around(0.5)
                .box(),
            selector=selectortools.Selector().by_leaves(),
        ),
    accent_and_staccato=consort.AttachmentExpression(
        attachments=(
            indicatortools.Articulation('accent'),
            indicatortools.Articulation('staccato'),
        ),
    ),
    selector=selectortools.Selector().by_leaves().by_logical_tie(pitched=True)
        [0],
),
    pitch_handler=consort.PitchClassPitchHandler(
        pitch_specifier='a c b d b f gs e d f cs',
        register_specifier=consort.RegisterSpecifier(),
        register_spread=0,
    ),
    rhythm_maker=zaira.materials.stuttering_rhythm_maker,
)
```

B.2.58  `zaira.materials.wind_trills_music_specifier`
from abjad.tools import spannertools
import consort
import zaira

wind_trills_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        trill_spanner=consort.AttachmentExpression(
            attachments=(
                spannertools.ComplexTrillSpanner(interval='+P4'),
                spannertools.ComplexTrillSpanner(interval='+P4'),
                spannertools.ComplexTrillSpanner(interval='+m3'),
                spannertools.ComplexTrillSpanner(interval='+m3'),
                spannertools.ComplexTrillSpanner(interval='+P4'),
                spannertools.ComplexTrillSpanner(interval='+m3'),
            ),
            selector=selectortools.select_pitched_runs(),
        ),
    ),
    pitch_handler=consort.PitchClassPitchHandler(
        pitch_specifier='c ef d',
        register_specifier=consort/RegisterSpecifier(
            base_pitch="g'",
        ),
    ),
    rhythm_maker=zaira.materials.sustained_rhythm_maker,
)

B.2.59  ZAIRA.MATERIALS.WOOD_AGITATION_MUSIC_SPECIFIER

# -*- encoding: utf-8 -*-
from abjad.tools import indicatortools
from abjad.tools import pitchtools
from abjad.tools import selectortools
from abjad.tools import spannertools
from abjad.tools import topleveltools
import new
import consort
import zaira

wood_agitation_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.erratic_dynamic_attachment_expression,
        stem_tremolo_spanner=consort.AttachmentExpression(
            attachments=spannertools.StemTremoloSpanner(),
            selector=selectortools.Selector(
                ).by_logical_tie(pitched=True
                ).by_duration('>', (1, 16))
            ),
        ),
    accent=consort.AttachmentExpression(
        attachments=(
            indicatortools.Articulation('accent'),
        ),
    ),
selector=selectortools.Selector(
    ).by_logical_tie(pitched=True)[0],
),
pitch_handler=consort.AbsolutePitchHandler(
pitch_specifier=pitchtools.PitchSegment(
    items=(
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.GUERO,
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.GUERO,
        zaira.makers.Percussion.GUERO,
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.GUERO,
        zaira.makers.Percussion.GUERO,
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.TAMBOURINE,
        zaira.makers.Percussion.TAMBOURINE,
       ),
       ),
   ),
),
rhythm_maker=new(
   zaira.materials.reiterating_rhythm_maker,
   denominators=(16, 4),
   extra_counts_per_division=(0, 1, 0, 1, 2),
   ),
)

B.2.60 zaira.materials.wood_bamboo_musicSpecifier

# -*- encoding: utf-8 -*-
from abjad.tools import pitchtools
from abjad.tools import indicatortools
from abjad.tools import selectortools
from abjad.tools import spannertools
import consort
import zaira

wood_bamboo_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=zaira.materials.midground_dynamic_attachment_expression,
        stem_tremolo_spanner=consort.AttachmentExpression(
            attachments=spannertools.StemTremoloSpanner(),
        ),
    ),
### SEGMENT MAKER

```python
segment_maker = zaira.makers.ZairaSegmentMaker(
    discard_final_silence=True,
    permitted_time_signatures=(
        (2, 4),
        (3, 8),
    ),
    tempo=indicatortools.Tempo(((1, 4), 72),
)
```

```python
ratio = mathtools.NonreducedRatio([2])
```

```python
segment_maker.desired_duration_in_seconds = (durationtools.Multiplier(sum(ratio), 91) *
zaira.materials.total_duration_in_seconds)
```
segment_maker.add_setting(
    timespan_maker=consort.FloodedTimespanMaker(),
    piano_rh=new(
        zaira.materials.piano_fanfare_music_specifier,
        pitch_handler__registerSpecifier__basePitch="g'",
    ),
    piano_lh=new(
        zaira.materials.piano_fanfare_music_specifier,
        pitch_handler__logical_tie_expressions=
            zaira.materials.piano_fanfare_music_specifier
            .pitch_handler.logical_tie_expressions[:1],
        pitch_handler__pitchSpecifier="g c a f d f e e'",
        pitch_handler__registerSpecifier__basePitch="g,,",
    ),
    drums=new(
        zaira.materials.percussion_fanfare_music_specifier,
        pitch_handler__pitchSpecifier=zaira.makers.Percussion.KICK_DRUM,
    ),
    metals=new(
        zaira.materials.percussion_fanfare_music_specifier,
        pitch_handler__pitchSpecifier=zaira.makers.Percussion.BRAKE_DRUM,
    ),
)

### DEPENDENT MUSIC SETTINGS ##################################################

segment_maker.add_setting(
    timespan_maker=zaira.materials.pedals_timespan_maker,
    piano_pedals=zaira.materials.piano_pedals_music_specifier,
)

B.3.2 zaira.segments.segment_B

# -*- encoding: utf-8 -*-
from abjad import new
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import mathtools
from abjad.tools import timespantools

import consort
import zaira

### SEGMENT MAKER ################################################################

segment_maker = zaira.makers.ZairaSegmentMaker(
    tempo=indicatortools.Tempo((1, 4), 48),
)
```python
ratio = mathtools.NonreducedRatio([1, 6, 3])
segment_maker.desired_duration_in_seconds = (durationtools.Multiplier(sum(ratio), 91) *
zaira.materials.total_duration_in_seconds)
fanfare_duration = durationtools.Duration(1, 16)

### WINDS SETTINGS

segment_maker.add_setting(
timespan_maker=zaira.materials.dense_timespan_maker,
timespan_identifier=consort.RatioPartsExpression(
    parts=(0, 2, 4),
    ratio=(1, 1, 1, 1, 1),
    mask_timespan=timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
),
clarinet=zaira.materials.wind_keyclick_music_specifier,
flute=zaira.materials.wind_keyclick_music_specifier,
oboe=zaira.materials.wind_keyclick_music_specifier,
)

segment_maker.add_setting(
timespan_maker=zaira.materials.dense_timespan_maker,
timespan_identifier=consort.RatioPartsExpression(
    parts=(1, 3),
    ratio=(3, 1, 2, 1, 1),
    mask_timespan=timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
),
oboe=zaira.materials.oboe_solo_music_specifier,
)

### PERCUSSION SETTINGS

segment_maker.add_setting(
timespan_maker=new(zaira.materials.dense_timespan_maker,
    reflect=True),
drums=zaira.materials.drum_tranquilo_music_specifier,
metals=zaira.materials.metal_tranquilo_music_specifier,
)
```
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        fuse_groups=True,
        padding=durationtools.Duration(1, 4),
    ),
    drums=zaira.materials.drum_brushed_musicSpecifier,
    metals=zaira.materials.metal_brushed_musicSpecifier,
)

### PIANO SETTINGS #

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        playing_groupings=(1,),
        playing_talea__counts=(5, 3, 3, 6, 4, 3),
        timespanSpecifier=consort.TimespanSpecifier(
            minimum_duration=0,
        ),
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3, 5),
        ratio=(1, 2, 1, 2, 1, 2, 1),
    ),
    piano_rh=new(
        zaira.materials.piano_flourish_musicSpecifier,
        pitch_handler__registerSpecifier__base_pitch="c'"',
    ),
    piano_lh=new(
        zaira.materials.piano_flourish_musicSpecifier,
        pitch_handler__registerSpecifier__base_pitch="c,",
    ),
)

### STRING SETTINGS #

segment_maker.add_setting(
    timespan_maker=zaira.materials.dense_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(0, 2, 4, 6),
        ratio=(1, 2, 1, 2, 1, 2, 1),
    ),
    piano_rh=zaira.materials.piano_prepared_treble_musicSpecifier,
    piano_lh=zaira.materials.piano_prepared_bass_musicSpecifier,
)

### STRING SETTINGS #

segment_maker.add_setting(
    timespan_maker=new(

zaira.materials.sparse_timespan_maker,
padding=durationtools.Duration(1, 4),
playing_groupings=(1,),
playing_talea_counts=(4, 3, 2, 4, 3),
reflect=True,
),
timespan_identifier=timespantools.Timespan(
    start_offset=fanfare_duration,
),

violin=new(
    zaira.materials.string_flourish_music_specifier,
    pitch_handler__registerSpecifier__base_pitch=\'None,\'
),

viola=new(
    zaira.materials.string_flourish_music_specifier,
    pitch_handler__registerSpecifier__base_pitch=\'c',
),

cello=new(
    zaira.materials.string_flourish_music_specifier,
    pitch_handler__registerSpecifier__base_pitch=\'c,\'
),

### FANFARE SETTINGS ####################################################################

segment_maker.add_setting(
    timespan_maker=consort.FloodedTimespanMaker(),
    timespan_identifier=timespantools.Timespan(
        stop_offset=fanfare_duration,
    ),
    piano_rh=new(
        zaira.materials.piano_fanfare_music_specifier,
        pitch_handler__registerSpecifier__base_pitch=\"g\",\n    ),
    piano_lh=new(
        zaira.materials.piano_fanfare_music_specifier,
        pitch_handler__logical_tie_expressions=\n            zaira.materials.piano_fanfare_music_specifier
                .pitch_handler.logical_tie_expressions[:-1],
            pitch_handler__pitch_specifier=\"g c a f d f e b e\",\n            pitch_handler__registerSpecifier__base_pitch=\"g,,\",\n    ),
    drums=new(
        zaira.materials.percussion_fanfare_music_specifier,
        pitch_handler__pitch_specifier=zaira.makers.Percussion.KICK_DRUM,
    ),
    metals=new(
        zaira.materials.percussion_fanfare_music_specifier,
        pitch_handler__pitch_specifier=zaira.makers.Percussion.BRAKE_DRUM,
### DEPENDENT MUSIC SETTINGS

```python
    segment_maker.add_setting(
        timespan_maker=zaira.materials.pedals_timespan_maker,
        piano_pedals=zaira.materials.piano_pedals_musicSpecifier,
    )
```

B.3.3  Zaira.segments.segment_c

```python
# -*- encoding: utf-8 -*-
from abjad import new
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import mathtools
from abjad.tools import pitchtools
from abjad.tools import timespantools
import consort
import zaira

### SEGMENT MAKER

```python
    segment_maker = zaira.makers.ZairaSegmentMaker(
        tempo=indicatortools.Tempo((1, 4), 72),
    )
```

```python
    ratio = mathtools.NonreducedRatio([7])
    segment_maker.desired_duration_in_seconds = (  
        durationtools.Multiplier(sum(ratio), 91) *  
        zaira.materials.total_duration_in_seconds  
    )
```

```python
    fanfare_duration = durationtools.Duration(1, 4)
```

### WINDS SETTINGS

```python
    segment_maker.add_setting(
        timespan_maker=zaira.materials.granular_timespan_maker,
        timespan_identifier=timespantools.Timespan(  
            start_offset=fanfare_duration,  
        ),
        flute=new(  
            zaira.materials.wind_trills_musicSpecifier,  
            pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(  
                operators=[
```
    pitchtools.Transposition(-3),
    pitchtools.Inversion(),
    ),
    seed=0,
    ),
  oboe=new(
    zaira.materials.wind_trills_musicSpecifier,
    pitch_handler__pitch_operationSpecifier=pitchtools.PitchOperation(
      operators=(
        pitchtools.Transposition(-3),
        pitchtools.Inversion(),
      ),
      seed=1,
    ),
  clarinet=new(
    zaira.materials.wind_trills_musicSpecifier,
    pitch_handler__pitch_operationSpecifier=pitchtools.PitchOperation(
      operators=(
        pitchtools.Transposition(-3),
        pitchtools.Inversion(),
      ),
      seed=2,
    ),
  )
}

    segment_maker.add_setting(
      timespan_maker=zaira.materials.sparse_timespan_maker,
      timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration,
      ),
    clarinet=new(
      zaira.materials.wind_slap_musicSpecifier,
      pitch_handler__registerSpecifier__base_pitch='D3',
    ),
    flute=zaira.materials.wind_slap_musicSpecifier,
    oboe=new(
      zaira.materials.wind_slap_musicSpecifier,
      pitch_handler__registerSpecifier__base_pitch='Bb3',
    ),
  )
}

### PERCUSSION SETTINGS ####################################################################

    segment_maker.add_setting(
      timespan_maker=zaira.materials.dense_timespan_maker,
      timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration,
      ),
drums=zaira.materials.drum_tranquilo_music_specifier, 
woods=zaira.materials.wood_agitation_music_specifier, 
)

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker, 
        reflect=True, 
    ), 
    timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration, 
    ), 
    drums=(
        zaira.materials.drum_brushed_music_specifier, 
        zaira.materials.drum_agitation_music_specifier, 
    ), 
)

### PIANO SETTINGS ##################################################################

segment_maker.add_setting(
    timespan_maker=zaira.materials.sparse_timespan_maker, 
    piano_rh=new(
        zaira.materials.piano_clusters_music_specifier, 
        pitch_handler__register_specifier__base_pitch="g'", 
    ), 
    piano_lh=new(
        zaira.materials.piano_clusters_music_specifier, 
        pitch_handler__register_specifier__base_pitch="g,,", 
    ), 
)

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker, 
        playing_groupings=(1,), 
        playing_talea_counts=(5, 3, 3, 3, 6, 4, 3), 
        timespan_specifier=consort.TimespanSpecifier(
            minimum_duration=0, 
        ), 
    ), 
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3, 5), 
        ratio=(1, 2, 1, 2, 1, 2, 1), 
    ), 
    piano_rh=new(
        zaira.materials.piano_flourish_music_specifier, 
        pitch_handler__register_specifier__base_pitch="c''", 
        pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
            operators=(
    …
pitchtools.Inversion(),
pitchtools.Transposition(3),
),
),
piano_lh=new(
zaira.materials.piano_flourish_music_specifier,
pitch_handler__register_specifier__base_pitch="c",
pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
operators=(
pitchtools.Inversion(),
pitchtools.Transposition(3),
),
),
)

## STRING SETTINGS ####################################################################

segment_maker.add_setting(
timespan_maker=new(
zaira.materials.sparse_timespan_maker,
 fuse_groups=True,
padding=durationtools.Duration(1, 4),
 reflect=True,
),
)

timespan_identifier=timespantools.Timespan(
 start_offset=fanfare_duration,
),

violin=new(
zaira.materials.string_flourish_music_specifier,
pitch_handler__register_specifier__base_pitch=None,
pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
 operators=(
pitchtools.Transposition(3),
),
),
rhythm_maker__talea__denominator=16,
seed=0,
),
viola=new(
zaira.materials.string_flourish_musicSpecifier,
pitch_handler__register_specifier__base_pitch='c',
pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
    operators=(
pitchtools.Transposition(3),
),
    rhythm_maker__talea__denominator=16,
    seed=1,
),
cello=new(
zaira.materials.string_flourish_musicSpecifier,
pitch_handler__register_specifier__base_pitch='c,',
pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
    operators=(
pitchtools.Transposition(3),
),
    rhythm_maker__talea__denominator=16,
    seed=2,
),
segment_maker.add_setting(
timespan_maker=new(
zaira.materials.tutti_timespan_maker,
padding=durationtools.Duration(1, 4),
playing_groupings=(1,),
    reflect=True,
),
timespan_identifier=consort.RatioPartsExpression(
    parts=(0, 2, 4),
    ratio=(1, 2, 1, 1, 1),
),
violin=new(
zaira.materials.string_undergrowth_musicSpecifier,
pitch_handler__register_specifier__base_pitch='g',
    seed=0,
),
viola=new(
zaira.materials.string_undergrowth_musicSpecifier,
pitch_handler__register_specifier__base_pitch='c',
    seed=1,
),
cello=new(
zaira.materials.string_undergrowth_musicSpecifier,
pitch_handler__register_specifier__base_pitch='c,',
    seed=2,
segment_maker.add_setting(
    timespan_maker=zaira.materials.dense_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3),
        ratio=(3, 1, 2, 1, 1),
        mask_timespan=timespantools.Timespan(
            start_offset=fanfare_duration,
        ),
        cello=zaira.materials.cello_solo_music_specifier,
    )
)

### SHAKER SETTINGS ###################################################################

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        padding=durationtools.Duration(3, 8),
    ),
    timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
    clarinet=zaira.materials.brazil_nut_musicSpecifier,
    flute=zaira.materials.brazil_nut_music_specifier,
    woods=zaira.materials.wood_bamboo_music_specifier,
    violin=zaira.materials.brazil_nut_music_specifier,
    viola=zaira.materials.brazil_nut_music_specifier,
)

### FANFARE SETTINGS ###################################################################

segmentMaker.add_setting(
    timespanMaker=consort.FloodedTimespanMaker(),
    timespanIdentifier=timespantools.Timespan(
        stopOffset=fanfare_duration,
    ),
    piano_rh=new(
        zaira.materials.piano_fanfare_music_specifier,
        pitch_handler__registerSpecifier__basePitch="g'",
    ),
    piano_lh=new(
        zaira.materials.piano_fanfare_music_specifier,
        pitch_handler__logical_tie_expressions=
            zaira.materials.piano_fanfare_music_specifier
                .pitch_handler.logical_tie_expressions[:1],
        pitch_handler__pitchSpecifier="g c a f d f e b e",
    )
pitch_handler__register_specifier__base_pitch="g,",
),
drums=new(
  zaira.materials.percussion_fanfare_music_specifier,
pitch_handler__pitch_specifier=zaira.makers.Percussion.KICK_DRUM,
),
metals=new(
  zaira.materials.percussion_fanfare_music_specifier,
pitch_handler__pitch_specifier=zaira.makers.Percussion.BRAKE_DRUM,
)
)

### DEPENDENT MUSIC SETTINGS ##################################################

B.3.4  zaира.segments.segment_d

# -*- encoding: utf-8 -*-
from abjad import new
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import mathtools
from abjad.tools import pitchtools
from abjad.tools import timespantools
import consort
import zaира

### SEGMENT MAKER ################################################################

segment_maker = zaира.makers.ZairaSegmentMaker(
  tempo=indicatortools.Tempo((1, 4), 84),
)

ratio = mathtools.NonreducedRatio([1, 2, 1])

durationtools.Multiplier(sum(ratio), 91) *
zaира.materials.total_duration_in_seconds

fanfare_duration = durationtools.Duration(1, 16)

### WIND SETTINGS ################################################################
segment_maker.add_setting(
    timespan_maker=zaira.materials.granular_timespan_maker,
    timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
    flute=new(
        zaira.materials.wind_trills_musicSpecifier,
        pitch_handler.pitch_operation_specifier=pitchtools.PitchOperation(
            pitchtools.Inversion(),
        ),
        seed=0,
    ),
    oboe=new(
        zaira.materials.wind_trills_musicSpecifier,
        pitch_handler.pitch_operation_specifier=pitchtools.PitchOperation(
            pitchtools.Inversion(),
        ),
        seed=1,
    ),
    clarinet=new(
        zaira.materials.wind_trills_musicSpecifier,
        pitch_handler.pitch_operation_specifier=pitchtools.PitchOperation(
            pitchtools.Inversion(),
        ),
        seed=2,
    ),
)

segment_maker.add_setting(
    timespan_maker=zaira.materials.dense_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3),
        ratio=(3, 1, 2, 1, 1),
        mask_timespan=timespantools.Timespan(
            start_offset=fanfare_duration,
        ),
    ),
    oboe=new(
        zaira.materials.oboe_solo_musicSpecifier,
        seed=1,
    ),
)

### PERCUSSION SETTINGS ###################################################################

segment_maker.add_setting(
    timespan_maker=zaira.materials.sparse_timespan_maker,
    metals=zaira.materials.metal_agitation_musicSpecifier,
    woods=zaira.materials.wood_agitation_musicSpecifier,
)
### PIANO SETTINGS

```python
segment_maker.add_setting(
    timespan_maker=new(zaira.materials.dense_timespan_maker,
    fuse_groups=True,
    ),
    piano_rh=zaira.materials.piano_drone_music_specifier,
)

segment_maker.add_setting(
    timespan_maker=zaira.materials.dense_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(0, 2, 4),
        ratio=(1, 2, 1, 2, 1),
    ),
    piano_rh=zaira.materials.piano_prepared_treble_musicSpecifier,
    piano_lh=zaira.materials.piano_prepared_bass_musicSpecifier,
)
```

### STRING SETTINGS

```python
segment_maker.add_setting(
    timespan_maker=zaira.materials.granular_timespan_maker,
    reflect=True,
    ),
    timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
    violin=new(zaira.materials.string_trills_musicSpecifier,
        pitch_handler__registerSpecifier__base_pitch="c'",
        pitch_handler__pitchOperationSpecifier=pitchtools.PitchOperation(
            operators=(
                pitchtools.Transposition(-3),
                pitchtools.Inversion(),
            ),
        ),
        seed=0,
    ),
    viola=new(zaira.materials.string_trills_musicSpecifier,
        pitch_handler__registerSpecifier__base_pitch="c'",
        pitch_handler__pitchOperationSpecifier=pitchtools.PitchOperation(
            operators=(
                pitchtools.Transposition(-3),
                pitchtools.Inversion(),
            ),
        ),
    )
```
cello='new('  
zaira.materials.string_trills_music_specifier,
pitch_handler__register_specifier__base_pitch='c',
pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(  
    operators=(
        pitchtools.Transposition(-3),
        pitchtools.Inversion(),
    ),
    seed=2,
),
)
}

segment_maker.add_setting(  
    timespan_maker='new('  
zaira.materials.sparse_timespan_maker,
padding=durationtools.Duration(1, 4),
    reflect=True,
),
    timespan_identifier=timespantools.Timespan(  
        start_offset=fanfare_duration,
    ),
    violin='new('  
zaira.materials.string_flourish_music_specifier,
pitch_handler__register_specifier__base_pitch=None,
pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(  
        pitchtools.Transposition(-3),
    ),
    rhythm_maker__talea__denominator=16,
    seed=0,
),
    viola='new('  
zaira.materials.string_flourish_music_specifier,
pitch_handler__register_specifier__base_pitch='c',
pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(  
        pitchtools.Transposition(-3),
    ),
    rhythm_maker__talea__denominator=16,
    seed=1,
),
    cello='new('  
zaira.materials.string_flourish_music_specifier,
pitch_handler__register_specifier__base_pitch='c',
pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(  
        pitchtools.Transposition(-3),
    ),
    rhythm_maker__talea__denominator=16,
    seed=2,
),
)
### FANFARE SETTINGS

```python
segment_maker.add_setting(
    timespan_maker=zaira.materials.sparse_timespan_maker,
    piano_rh=new(
        zaira.materials.piano_clusters_musicSpecifier,
        pitch_handler__register_specifier__base_pitch="c'",
    ),
    piano_lh=new(
        zaira.materials.piano_clusters_musicSpecifier,
        pitch_handler__register_specifier__base_pitch="c,",
    ),
)
segment_maker.add_setting(
    timespan_maker=consort.FloodedTimespanMaker(),
    timespan_identifier=timespanTools.Timespan(
        stop_offset=fanfare_duration,
    ),
    piano_rh=new(
        zaira.materials.piano_fanfare_music_specifier,
        pitch_handler__register_specifier__base_pitch="g'",
    ),
    piano_lh=new(
        zaira.materials.piano_fanfare_music_specifier,
        pitch_handler__logical_tie_expressions=,
            zaira.materials.piano_fanfare_musicSpecifier
            pitch_handler.logical_tie_expressions[::1],
        pitch_handler__pitch_specifier="g c a f d f e b e",
        pitch_handler__register_specifier__base_pitch="g,,",
    ),
    drums=new(
        zaira.materials.percussion_fanfare_music_specifier,
        pitch_handler__pitch_specifier=zaira.makers.Percussion.KICK_DRUM,
    ),
    metals=new(
        zaira.materials.percussion_fanfare_music_specifier,
        pitch_handler__pitch_specifier=zaira.makers.Percussion.BRAKE_DRUM,
    ),
)
```

### DEPENDENT MUSIC SETTING

```python
segment_maker.add_setting(
    timespan_maker=zaira.materials.pedals_timespan_maker,
    piano_pedals=zaira.materials.piano_pedals_musicSpecifier,
)
```
### SEGMENT MAKER

```python
segment_maker = zaira.makers.ZairaSegmentMaker(
    tempo=indicatortools.Tempo((1, 4), 72),
)
```

```python
ratio = mathtools.NonreducedRatio([1, 8, 4])
```

```python
segment_maker.desired_duration_in_seconds = (durationtools.Multiplier(sum(ratio), 91) *
zaira.materials.total_duration_in_seconds)
```

```python
fanfare_duration = durationtools.Duration(3, 4)
```

### WINDS SETTINGS

```python
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.granular_timespan_maker,
        reflect=True,
    ),
    timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
    flute=new(
        zaira.materials.wind_trills_musicSpecifier,
        seed=0,
    ),
    oboe=new(
        zaira.materials.wind_trills_musicSpecifier,
        seed=1,
    ),
    clarinet=new(
        zaira.materials.wind_trills_musicSpecifier,
        seed=2,
    ),
)
```
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.dense_timespan_maker,
        playing_groupings=({1,}),
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=({1, 3, 5}),
        ratio=({1, 2, 1, 2, 1, 2, 1}),
        mask_timespan=timespan_tools.Timespan(
            start_offset=fanfare_duration,
        ),
    ),
    clarinet=new(
        zaira.materials.w wind_airtone_music_specifier, 
        pitch_handler__register_specifier__base_pitch='D3', 
    ),
    flute=zaira.materials.wind_airtone_music_specifier, 
    oboe=new(
        zaira.materials.wind_airtone_music_specifier, 
        pitch_handler__register_specifier__base_pitch='Bb3', 
    ),
)
segment_maker.add_setting(
    timespan_maker=zaira.materials.dense_timespan_maker, 
    timespan_identifier=consort.RatioPartsExpression(
        parts=({0, 2, 4, 6}),
        ratio=({1, 1, 3, 1, 1, 2}),
        mask_timespan=timespan_tools.Timespan(
            start_offset=fanfare_duration,
        ),
    ),
    clarinet=zaira.materials.wind_keyclick_music_specifier, 
    flute=zaira.materials.wind_keyclick_music_specifier, 
    oboe=zaira.materials.wind_keyclick_music_specifier, 
)
segment_maker.add_setting(
    timespan_maker=zaira.materials.sparse_timespan_maker, 
    timespan_identifier=timespan_tools.Timespan( 
        start_offset=fanfare_duration, 
    ),
    clarinet=new(
        zaira.materials.wind_slap_music_specifier, 
        pitch_handler__register_specifier__base_pitch='D3', 
    ),
    flute=zaira.materials.wind_slap_music_specifier, 
    oboe=new(
        zaira.materials.wind_slap_music_specifier, 
        pitch_handler__register_specifier__base_pitch='Bb3', 
    ),
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.dense_timespan_maker,
        reflect=True,
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(0, 2, 4),
        ratio=(1, 1, 1, 1),
        mask_timespan=timespantools.Timespan(
            start_offset=fanfare_duration,
        ),
        oboe=zaira.materials.oboe_solo_musicSpecifier,
    )
)

### PERCUSSION SETTINGS #####################################################################

segment_maker.add_setting(
    timespan_maker=zaira.materials.dense_timespan_maker,
    drums=zaira.materials.drum_agitation_musicSpecifier,
    metals=zaira.materials.metal_agitation_musicSpecifier,
)

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        fuse_groups=True,
        padding=durationtools.Duration(1, 4),
        reflect=True,
    ),
    drums=zaira.materials.drum_brushed_musicSpecifier,
    metals=zaira.materials.metal_brushed_musicSpecifier,
)

### PIANO SETTINGS #####################################################################

segment_maker.add_setting(
    timespan_maker=zaira.materials.dense_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(0, 2, 4, 6),
        ratio=(1, 2, 1, 2, 1, 2, 1),
    ),
    piano_rh=zaira.materials.piano_prepared_treble_musicSpecifier,
    piano_lh=zaira.materials.piano_prepared_bass_musicSpecifier,
)

800
segment_maker.add_setting(
    timespan_maker=zaira.materials.sparse_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1,),
        ratio=ratio,
    ),
    piano_rh=new(
        zaira.materials.piano_clusters_musicSpecifier,
        pitch_handler__register_specifier__base_pitch="g''",
    ),
    piano_lh=new(
        zaira.materials.piano_clusters_musicSpecifier,
        pitch_handler__register_specifier__base_pitch="c,",
    ),
)

segment_maker.add_setting(
    timespan_maker=zaira.materials.dense_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(2,),
        ratio=ratio,
    ),
    piano_rh=new(
        zaira.materials.piano_clusters_musicSpecifier,
        attachment_handler__dynamic_expression=zaira.materials.erratic_dynamic_attachment_expression,
        pitch_handler__register_specifier__base_pitch="g",
    ),
    piano_lh=new(
        zaira.materials.piano_clusters_musicSpecifier,
        attachment_handler__dynamic_expression=zaira.materials.erratic_dynamic_attachment_expression,
        pitch_handler__register_specifier__base_pitch="c,",
    ),
)

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        fuse_groups=True,
        padding=durationtools.Duration(1, 4),
        reflect=True,
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(0, 2, 4),
        ratio=(1, 2, 1, 2, 1),
    ),
    piano_rh=zaira.materials.piano_guero_musicSpecifier,
    piano_lh=new(
        zaira.materials.piano_guero_musicSpecifier,
        seed=1,
    ),
## STRING SETTINGS

```python
segment_maker.add_setting(
    timespan_maker=zaira.materials.granular_timespan_maker,
    timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
    violin=new(
        zaira.materials.string_trills_musicSpecifier,
        pitch_handler__register_specifier__base_pitch="c'",
        pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
            pitchtools.Transposition(-3),
        ),
        seed=0,
    ),
    viola=new(
        zaira.materials.string_trills_musicSpecifier,
        pitch_handler__register_specifier__base_pitch="c'",
        pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
            pitchtools.Transposition(-3),
        ),
        seed=1,
    ),
    cello=new(
        zaira.materials.string_trills_musicSpecifier,
        pitch_handler__register_specifier__base_pitch='c',
        pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
            pitchtools.Transposition(-3),
        ),
        seed=2,
    ),
)

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.tutti_timespan_maker,
        padding=durationtools.Duration(1, 4),
        playing_groupings=(1,),
        reflect=True,
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(0, 2, 4, 6),
        ratio=(1, 2, 1, 1, 1, 2, 1),
        mask_timespan=timespantools.Timespan(
            start_offset=fanfare_duration,
        ),
    ),
    violin=new(
        zaira.materials.string_undergrowth_musicSpecifier,
    ),
)```
    pitch_handler__register_specifier__base_pitch='g',
    seed=0,
),
  ),
  viola=new(
    zaira.materials.string_undergrowth_musicSpecifier,
    pitch_handler__register_specifier__base_pitch='c',
    seed=1,
  ),
  cello=new(
    zaira.materials.string_undergrowth_musicSpecifier,
    pitch_handler__register_specifier__base_pitch='c,',
    seed=2,
  ),
)

segment_maker.add_setting(
  timespan_maker=new(
    zaira.materials.tutti_timespan_maker,
    padding=durationtools.Duration(1, 4),
    playing_groupings=(1,),
    playing_talea_counts=(8,),
    reflect=True,
    repeat=False,
  ),
  timespan_identifier=consort.RatioPartsExpression(
    parts=(1, 3, 5),
    ratio=(1, 2, 1, 1, 1, 1, 2, 1),
    mask_timespan=timespantools.Timespan(
      start_offset=fanfare_duration,
    ),
  ),
  violin=new(
    zaira.materials.string_tutti_overpressure_musicSpecifier,
    pitch_handler__pitchSpecifier='g',
    seed=0,
  ),
  viola=new(
    zaira.materials.string_tutti_overpressure_musicSpecifier,
    pitch_handler__pitchSpecifier='c',
    seed=1,
  ),
  cello=new(
    zaira.materials.string_tutti_overpressure_musicSpecifier,
    pitch_handler__pitchSpecifier='c,',
    seed=2,
  ),
)

### FANFARE SETTINGS ##########################################################

segment_maker.add_setting(

timespan_maker=consort.FloodedTimespanMaker(),
timespan_identifier=timespantools.Timespan(
    stop_offset=fanfare_duration,
),
piano_rh=new(
    zaira.materials.piano_fanfare_musicSpecifier,
    pitch_handler__registerSpecifier__base_pitch="g'",
),
piano_lh=new(
    zaira.materials.piano_fanfare_musicSpecifier,
    pitch_handler__logicalTieExpressions=*
        zaira.materials.piano_fanfare_musicSpecifier
        .pitch_handler.logicalTieExpressions[::1],
    pitch_handler__pitchSpecifier="g c a f d f e",
    pitch_handler__registerSpecifier__base_pitch="g,,",
),
drums=new(
    zaira.materials.percussion_fanfare_musicSpecifier,
    pitch_handler__pitchSpecifier=*
        zaira.materials.percussion_fanfare_musicSpecifier
        .pitch_handler.pitchSpecifier=*
        zaira.materials.percussion_fanfare_musicSpecifier
        .pitch_handler.logicalTieExpressions[::1],
    pitch_handler__pitchSpecifier="g c a f d f e",
    pitch_handler__registerSpecifier__base_pitch="g,,",
),
metals=new(
    zaira.materials.percussion_fanfare_musicSpecifier,
    pitch_handler__pitchSpecifier=*
        zaira.materials.percussion_fanfare_musicSpecifier
        .pitch_handler.pitchSpecifier=*
        zaira.materials.percussion_fanfare_musicSpecifier
        .pitch_handler.logicalTieExpressions[::1],
    pitch_handler__pitchSpecifier="g c a f d f e",
    pitch_handler__registerSpecifier__base_pitch="g,,",
),

### DEPENDENT MUSIC SETTINGS

segment_maker.add_setting(
    timespan_maker=zaira.materials.pedals_timespan_maker,
    piano_pedals=zaira.materials.piano_pedals_musicSpecifier,
)
ratio = mathtools.NonreducedRatio([1, 14])

segment_maker.desired_duration_in_seconds = (durationtools.Multiplier(sum(ratio), 91) *
    zaira.materials.total_duration_in_seconds)

fanfare_duration = durationtools.Duration(1, 16),

### WINDS SETTINGS ####################################################################

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.dense_timespan_maker,
        playing_groupings=(1,),
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3, 5),
        ratio=(1, 1, 2, 1, 2, 1, 2),
        mask_timespan=timespantools.Timespan(
            start_offset=fanfare_duration,
        ),
    ),
    clarinet=new(
        zaira.materials.wind_airtone_musicSpecifier,
        pitch_handler__registerSpecifier__base_pitch='D3',
    ),
    flute=zaira.materials.wind_airtone_musicSpecifier,
    oboe=new(
        zaira.materials.wind_airtone_musicSpecifier,
        pitch_handler__registerSpecifier__base_pitch='Bb3',
    ),
)

segment_maker.add_setting(
    timespan_maker=zaira.materials.sparse_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(0, 2, 4),
        ratio=(3, 1, 2, 1, 1),
        mask_timespan=timespantools.Timespan(
            start_offset=fanfare_duration,
        ),
    ),
    clarinet=zaira.materials.wind_keyclick_musicSpecifier,
    flute=zaira.materials.wind_keyclick_musicSpecifier,
    oboe=zaira.materials.wind_keyclick_musicSpecifier,
)

segment_maker.add_setting
timespan_maker = new(
    zaira.materials.granular_timespan_maker,
    playing_talea_counts=(2, 2, 3, 2, 7, 1, 3, 2, 1),
),
timespan_identifier = consort.RatioPartsExpression(
    parts=(0, 2, 4),
    ratio=(1, 1, 1, 1),
    mask_timespan = timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
),
oboe = new(
    zaira.materials.oboe_solo_music_specifier,
    pitch_handler__pitch_operation_specifier = pitchtools.PitchOperation(
        pitchtools.Transposition(-3),
    ),
)

### PERCUSSION SETTINGS ######################################################################

segment_maker.add_setting(
    timespan_maker = new(
        zaira.materials.sustained_timespan_maker,
        fuse_groups=True,
    ),
    drums = new(
        zaira.materials.percussion_brushed_musicSpecifier,
        pitch_handler__pitch_specifier = zaira.makers.Percussion.BASS_DRUM,
    ),
    metals = new(
        zaira.materials.percussion_brushed_musicSpecifier,
        pitch_handler__pitch_specifier = zaira.makers.Percussion.TAM_TAM,
    ),
)

segment_maker.add_setting(
    timespan_maker = new(
        zaira.materials.sparse_timespan_maker,
        fuse_groups=True,
    ),
    timespan_identifier = consort.RatioPartsExpression(
        parts=(0, 2, 4),
        ratio=(1, 1, 1, 1),
    ),
    drums = new(
        zaira.materials.percussion_brushed_tremolo_musicSpecifier,
        pitch_handler__pitch_specifier = zaira.makers.Percussion.BASS_DRUM,
    ),
    metals = new(
        zaira.materials.percussion_brushed_tremolo_musicSpecifier,
    )
)
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.dense_timespan_maker,
        fuse_groups=True,
    ),
    piano_rh=zaira.materials.piano_drone_musicSpecifier,
)

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        fuse_groups=True,
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3),
        ratio=(1, 1, 1, 1),
    ),
    drums=new(
        zaira.materials.percussion_superball_musicSpecifier,
        pitch_handler__pitch_specifier=zaira.makers.Percussion.BASS_DRUM,
    ),
    metals=new(
        zaira.materials.percussion_superball_musicSpecifier,
        pitch_handler__pitch_specifier=zaira.makers.Percussion.TAM_TAM,
    )
)

### PIANO SETTINGS ####################################################################

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.dense_timespan_maker,
        fuse_groups=True,
    ),
    piano_rh=zaira.materials.piano_drone_musicSpecifier,
)

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        playing_groupings=(1,),
        playing_talea_counts=(3, 3, 3, 3, 5, 3, 4),
        timespanSpecifier=consort.TimespanSpecifier(
            minimum_duration=0,
        ),
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3, 5),
        ratio=(3, 1, 2, 1, 3, 1, 1),
    ),
    piano_rh=new(
        zaira.materials.piano_flourish_musicSpecifier,
        pitch_handler__register_specifier__base_pitch="c'",
        pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
pitchtools.Transposition(3),
    ),
    seed=1,
    ),
piano_lh=new(
    zaira.materials.piano_flourish_music_specifier,
    pitch_handler__register_specifier__base_pitch="c,",
    pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
        pitchtools.Transposition(3),
        ),
    seed=2,
    ),
)

### STRING SETTINGS ####################################################################

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.tutti_timespan_maker,
        padding=durationtools.Duration(1, 4),
        playing_groupings=(1,),
        reflect=True,
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(0, 2, 4, 6),
        ratio=(1, 2, 1, 1, 1, 2, 1),
        mask_timespan=timespantools.Timespan(
            start_offset=fanfare_duration,
        ),
    ),
    violin=new(
        zaira.materials.string_undergrowth_music_specifier,
        pitch_handler__register_specifier__base_pitch="g",
        seed=0,
    ),
    viola=new(
        zaira.materials.string_undergrowth_music_specifier,
        pitch_handler__register_specifier__base_pitch="c",
        seed=1,
    ),
    cello=new(
        zaira.materials.string_undergrowth_music_specifier,
        pitch_handler__register_specifier__base_pitch="c,'",
        seed=2,
    ),
)

segment_maker.add_setting(
    timespan_maker=zaira.materials.dense_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3),
    )
ratio=(3, 1, 2, 1, 1),
mask_timespan=timespantools.Timespan(
    start_offset=fanfare_duration,
),
cello=new(
    zaira.materials.cello_solo_musicSpecifier,
    seed=1,
),
### SHAKER SETTINGS   ###################################################################
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.tutti_timespan_maker,
        playing_talea_counts=(3, 2, 3, 2, 4),
        playing_groupings=(1,),
        padding=durationtools.Duration(3, 8),
    ),
    clarinet=zaira.materials.brazil_nut_musicSpecifier,
    flute=zaira.materials.brazil_nut_musicSpecifier,
    viola=zaira.materials.brazil_nut_musicSpecifier,
    violin=zaira.materials.brazil_nut_musicSpecifier,
)
### FANFARE SETTINGS   ##################################################################
segment_maker.add_setting(
    timespan_maker=consort.FloodedTimespanMaker(),
    timespan_identifier=timespantools.Timespan(
        stop_offset=fanfare_duration,
    ),
    piano_rh=new(
        zaira.materials.piano_fanfare_musicSpecifier,
        pitch_handler__registerSpecifier__base_pitch="g'",
    ),
    piano_lh=new(
        zaira.materials.piano_fanfare_musicSpecifier,
        pitch_handler__logical_tie_expressions=
            zaira.materials.piano_fanfare_musicSpecifier.
            pitch_handler.logical_tie_expressions[:-1],
        pitch_handler__pitchSpecifier="g c a f d f e b e",
        pitch_handler__registerSpecifier__base_pitch="g,",
    ),
    drums=new(
        zaira.materials.percussion_fanfare_musicSpecifier,
        pitch_handler__pitchSpecifier=zaira.makers.Percussion.KICK_DRUM,
    ),
    metals=new(
        zaira.materials.percussion_fanfare_musicSpecifier,
zaira.materials.percussion_fanfare_music_specifier,
pitch_handler__pitch_specifier=zaira.makers.Percussion.BRAKE_DRUM,
);

### DEPENDENT MUSIC SETTINGS

segment_maker.add_setting(
timespan_maker=zaira.materials.pedals_timespan_maker,
piano_pedals=zaira.materials.piano_pedals_music_specifier,
)

B.3.7  Zaira.segments.segment_f2

# -*- encoding: utf-8 -*-
from abjad import new
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import mathtools
from abjad.tools import pitchtools
import consort
import zaira

### SEGMENT MAKER

segment_maker = zaira.makers.ZairaSegmentMaker(
tempo=indicatortools.Tempo((1, 4), 48),
)

ratio = mathtools.NonreducedRatio([7])

segment_maker.desired_duration_in_seconds = (durationtools.Multiplier(sum(ratio), 91) * 
zaira.materials.total_duration_in_seconds)

### WINDS SETTINGS

segment_maker.add_setting(
timespan_maker=new(
zaira.materials.dense_timespan_maker,
playing_groupings=(1,),
silence_talea__denominator=4,
),
timespan_identifier=consort.RatioPartsExpression(
parts=(1, 3, 5),
ratio=(1, 1, 2, 1, 2, 1, 2),
),
)
clarinet = new(zaira.materials.wind_airtone_musicSpecifier,
pitch_handler__registerSpecifier__base_pitch = 'D3',
),
flute = zaira.materials.wind_airtone_musicSpecifier,

oboe = new(zaira.materials.wind_airtone_musicSpecifier,
pitch_handler__registerSpecifier__base_pitch = 'Bb3',
),

segment_maker.add_setting(

timespan_maker = zaira.materials.sparse_timespan_maker,
timespan_identifier = consort.RatioPartsExpression(
    parts = (0, 2, 4),
    ratio = (1, 2, 1, 2),
),
clarinet = zaira.materials.wind_keyclick_musicSpecifier,
flute = zaira.materials.wind_keyclick_musicSpecifier,

segment_maker.add_setting(

timespan_maker = new(zaira.materials.granular_timespan_maker,
    playing_talea_counts = (1, 1, 1, 3, 2, 1),
),
timespan_identifier = consort.RatioPartsExpression(
    parts = (1, 3),
    ratio = (2, 1, 2, 1, 1),
),

oboe = new(zaira.materials.oboe_solo_musicSpecifier,
pitch_handler__pitchOperationSpecifier = pitchtools.PitchOperation(
    pitchtools.Transposition(-5),
    ),
),
#

### PERCUSSION SETTINGS

segment_maker.add_setting(

timespan_maker = new(zaira.materials.sustained_timespan_maker,
    fuse_groups = True,
    timespanSpecifier = consort.TimespanSpecifier(
        minimum_duration = durationtools.Duration(1, 4),
    ),
),

drums = new(
zaira.materials.percussion_superball_musicSpecifier,
pitchHandler__pitchSpecifier=zaira.makers.Percussion.BASS_DRUM,
),
metals=new(zaira.materials.percussion_superball_musicSpecifier,
pitchHandler__pitchSpecifier=zaira.makers.Percussion.TAM_TAM,
),
)

segment_maker.add_setting(
timespan_maker=zaira.materials.sparse_timespan_maker,
timespan_identifier=consort.RatioPartsExpression(
  parts=(1, 3),
  ratio=(1, 1, 1, 1, 1),
),
drums=new(zaira.materials.percussion_brushed_musicSpecifier,
pitchHandler__pitchSpecifier=zaira.makers.Percussion.TAM_TAM,
),
metals=new(zaira.materials.percussion_brushed_musicSpecifier,
pitchHandler__pitchSpecifier=zaira.makers.Percussion.BASS_DRUM,
),
)

### PIANO SETTINGS ###################################################################

segment_maker.add_setting(
timespan_maker=new(zaira.materials.sustained_timespan_maker,
fuse_groups=True,
reflect=True,
),
piano_rh=new(zaira.materials.piano_drone_musicSpecifier,
pitchHandler__pitchSpecifier='b',
),
)

### STRING SETTINGS ###################################################################

segment_maker.add_setting(
timespan_maker=new(zaira.materials.tutti_timespan_maker,
padding=durationtools.Duration(1, 4),
playing_groupings=(1,),
reflect=True,
),
timespan_identifier=consort.RatioPartsExpression(
parts=(0, 2, 4),
ratio=(1, 2, 1, 3, 1),
)

violin=new(
  zaira.materials.string_undergrowth_musicSpecifier,
  pitch_handler__registerSpecifier__base_pitch='g',
  seed=0,
),

viola=new(
  zaira.materials.string_undergrowth_musicSpecifier,
  pitch_handler__registerSpecifier__base_pitch='c',
  seed=1,
),

cello=new(
  zaira.materials.string_undergrowth_musicSpecifier,
  pitch_handler__registerSpecifier__base_pitch='c,',
  seed=2,
)
)

segment_maker.add_setting(
  timespan_maker=new(
    zaira.materials.dense_timespan_maker,
    reflect=True,
  ),
  timespan_identifier=consort.RatioPartsExpression(
    parts=(0, 2, 4),
    ratio=(1, 1, 1, 1, 1),
  ),
  cello=zaira.materials.cello_solo_musicSpecifier,
)

### SHAKER SETTINGS ####################################################################

segment_maker.add_setting(
  timespan_maker=new(
    zaira.materials.tutti_timespan_maker,
    playing_talea_counts=(3, 2, 3, 2, 4),
    playing_groupings=(1,),
    padding=durationtools.Duration(3, 8),
  ),
  clarinet=zaira.materials.brazil_nut_musicSpecifier,
  flute=zaira.materials.brazil_nut_musicSpecifier,
  viola=zaira.materials.brazil_nut_musicSpecifier,
  violin=zaira.materials.brazil_nut_musicSpecifier,
)

### DEPENDENT MUSIC SETTINGS ####################################################################
import zaira

### SEGMENT MAKER

```python
segment_maker = zaira.makers.ZairaSegmentMaker(
    tempo=indicatortools.Tempo((1, 4), 96),
)
ratio = mathtools.NonreducedRatio([1, 4, 2])
segment_maker.desired_duration_in_seconds = (durationtools.Multiplier(sum(ratio), 91) *
zaira.materials.total_duration_in_seconds)
fanfare_duration = durationtools.Duration(13, 16)

### WINDS SETTINGS
```

```python
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.granular_timespan_maker,
        reflect=True,
    ),
    timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
    flute=new(
        zaira.materials.wind_trills_musicSpecifier,
        pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
            pitchtools.Inversion(),
        ),
        seed=0,
    ),
    oboe=new(
```
zaira.materials.wind_trills_musicSpecifier,
pitch_handler__pitch_operationSpecifier=pitchtools.PitchOperation(
    pitchtools.Inversion(),
),
seed=1,
),
clarinet=new(
    zaira.materials.wind_trills_musicSpecifier,
pitch_handler__pitch_operationSpecifier=pitchtools.PitchOperation(
    pitchtools.Inversion(),
),
seed=2,
),
}

### PERCUSSION SETTINGS #

segment_maker.add_setting(
timespan_maker=new(
    zaira.materials.dense_timespan_maker,
    reflect=True,
),
drums=zaira.materials.drum_storm_musicSpecifier,
)

segment_maker.add_setting(
timespan_maker=new(
    zaira.materials.sparse_timespan_maker,
),
drums=zaira.materials.drum_agitation_musicSpecifier,
)

segment_maker.add_setting(
timespan_music=zaira.materials.sparse_timespan_musicSpecifier,
timespan_identifier=consort.RatioPartsExpression(
    parts=(1, 3),
    ratio=(2, 1, 1, 1, 1),
),
metals=new(
    zaira.materials.percussion_reiteration_musicSpecifier,
pitch_handler__pitchSpecifier=zaira.makers.Percussion.BAMBOO_WINDCHIMES,
),
)

### PIANO SETTINGS #

segment_maker.add_setting(
timespan_music=zaira.materials.tutti_timespan_musicSpecifier,
piano_rh = new(zaira.materials.piano_clusters_musicSpecifier,
pitch_handler__registerSpecifier__base_pitch="g''",
),
piano_lh = new(zaira.materials.piano_clusters_musicSpecifier,
pitch_handler__registerSpecifier__base_pitch="g,",
seed=1,
),

### STRING SETTINGS ###################################################################

segment_maker.add_setting(
  timespan_maker=zaira.materials.granular_timespan_maker,
timespan_identifier=timespantools.Timespan(
    start_offset=fanfare_duration,
  ),
  violin=new(zaira.materials.string_trills_musicSpecifier,
pitch_handler__registerSpecifier__base_pitch="c'",
seed=0,
),
  viola=new(zaira.materials.string_trills_musicSpecifier,
pitch_handler__registerSpecifier__base_pitch='c',
seed=1,
),
  cello=new(zaira.materials.string_trills_musicSpecifier,
pitch_handler__registerSpecifier__base_pitch='c,',
seed=2,
)
)

segment_maker.add_setting(
  timespan_maker=new(zaira.materials.tutti_timespan_maker,
padding=durationtools.Duration(1, 4),
playing_groupings={(1,)},
playing_talea__counts={8,},
repeat=False,
),
timespan_identifier=consort.RatioPartsExpression(
  parts={(1, 3, 5),
  ratio={(1, 2, 1, 1, 2, 1),
  mask_timespan=timespantools.Timespan(
    start_offset=fanfare_duration,
  ),
  ),
  violin=new(}
zaira.materials.string_tutti_overpressure_musicSpecifier,
pitch_handler__pitchSpecifier='g',
seed=0,
)
viola=new(
zaira.materials.string_tutti_overpressure_musicSpecifier,
pitch_handler__pitchSpecifier='c',
seed=1,
)
cello=new(
zaira.materials.string_tutti_overpressure_musicSpecifier,
pitch_handler__pitchSpecifier='c,',
seed=2,
)
)

segment_maker.add_setting(
timespan_maker=new(
zaira.materials.granular_timespan_maker,
playing_talea__counts=(5, 4, 6, 5, 3, 5, 9),
),
timespan_identifier=consort.RatioPartsExpression(
parts=(1, 3, 5),
ratio=(1, 1, 1, 1, 1, 1),
mask_timespan=timespantools.Timespan(
    start_offset=fanfare_duration,
),
),
cello=new(
zaira.materials.cello_solo_musicSpecifier,
pitch_handler__pitchOperationSpecifier=pitchtools.PitchOperation(
    pitchtools.Transposition(3),
),
)

### FANFARE SETTINGS ###################################################################

segment_maker.add_setting(
timespan_maker=consort.FloodedTimespanMaker(),
timespan_identifier=timespantools.Timespan(
    stop_offset=fanfare_duration,
),
piano_rh=new(
zaira.materials.piano_fanfare_musicSpecifier,
pitch_handler__registerSpecifier__base_pitch="g"),
),
piano_lh=new(
zaira.materials.piano_fanfare_musicSpecifier,
pitch_handler__logical_tie_expressions=
    zaira.materials.piano_fanfare_musicSpecifier
.. _zaira.segments.segment_h:

## B.3.9  ZAIRA.SECONDS.SEGMENT_H

```python
# -*- encoding: utf-8 -*-

import consort

import zaira

### SEGMENT MAKER ###############################################################################

segment_maker = zaira.makers.ZairaSegmentMaker(
    tempo=indicatortools.Tempo((1, 4), 72),
)

ratio = mathtools.NonreducedRatio([5])

segment_maker.desired_duration_in_seconds = (durationtools.Multiplier(sum(ratio), 91) *
    zaira.materials.total_duration_in_seconds
)

fanfare_duration = durationtools.Duration(1, 8),
```
### WINDS SETTINGS

```python
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        playing_groupings=(1,),
        reflect=True,
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3, 5),
        ratio=(1, 2, 1, 2, 1, 2, 1),
        mask_timespan=timespantools.Timespan(
            start_offset=fanfare_duration,
        ),
    ),
    clarinet=new(
        zaira.materials.wind_airtone_musicSpecifier,
        pitch_handler__register_specifier__base_pitch='D3',
    ),
    flute=zaira.materials.wind_airtone_musicSpecifier,
    oboe=new(
        zaira.materials.wind_airtone_musicSpecifier,
        pitch_handler__register_specifier__base_pitch='Bb3',
    ),
)
```

```python
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        playing_groupings=(1,),
    ),
    timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
    clarinet=new(
        zaira.materials.wind_slap_musicSpecifier,
        pitch_handler__register_specifier__base_pitch='D3',
    ),
    flute=zaira.materials.wind_slap_musicSpecifier,
    oboe=new(
        zaira.materials.wind_slap_musicSpecifier,
        pitch_handler__register_specifier__base_pitch='Bb3',
    ),
)
```

```python
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        playing_groupings=(1,),
    ),
    timespan_identifier=consort.RatioPartsExpression(
```
### PERCUSSION SETTINGS

```python
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        fuse_groups=True,
        reflect=True,
    ),
    timespan_identifier=timespantools_TIMESpan(
        start_offset=fanfare_duration,
    ),
    drums=new(
        zaira.materials.drum_brushed_music_specifier,
        pitch_handler_pitch_specifier=zaira.makers.Percussion.BASS_DRUM,
    ),
)
```

### PIANO SETTINGS

```python
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        playing_groupings=(1,),
        playing_talea_counts=(5, 3, 3, 6, 4, 3),
        timespan_specifier=consort.TimespanSpecifier(
            minimum_duration=0,
        ),
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3, 5),
        ratio=(1, 2, 1, 2, 1, 2, 1),
        mask_timespan=timespantools_TIMESpan(
            start_offset=fanfare_duration,
        ),
    ),
    piano_rh=new(
        zaira.materials.piano_flourish_music_specifier,
        attachment_handler_dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
        pitch_handler_register_specifier_base_pitch="c'",
    ),
)
137 pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
138     operators=(
139         pitchtools.Inversion(),
140         pitchtools.Transposition(-3),
141     ),
142 ),
143 piano_lh=new(
144     zaira.materials.piano_flourish_musicSpecifier,
145     attachment_handler__dynamic_expression=zaira.materials.background_dynamic_attachment_expression,
146     pitch_handler__register_specifier__base_pitch="c",
147     pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
148         operators=(
149             pitchtools.Inversion(),
150             pitchtools.Transposition(-3),
151         ),
152     ),
153 ),
154 )
155 
156 segment_maker.add_setting(
157     timespan_maker=zaira.materials.sparse_timespan_maker,
158     timespan_identifier=timespantools.Timespan(
159         start_offset=fanfare_duration,
160     ),
161     piano_rh=new(
162         zaira.materials.piano_prepared_treble_musicSpecifier,
163         rhythm_maker=zaira.materials.sustained_rhythm_maker,
164     ),
165     piano_lh=new(
166         zaira.materials.piano_prepared_bass_musicSpecifier,
167         rhythm_maker=zaira.materials.sustained_rhythm_maker,
168     ),
169 )
170 
171 segment_maker.add_setting(
172     timespan_maker=zaira.materials.tutti_timespan_maker,
173     timespan_identifier=timespantools.Timespan(
174         start_offset=fanfare_duration,
175     ),
176     piano_rh=new(
177         zaira.materials.piano_clusters_musicSpecifier,
178         attachment_handler__dynamic_expression=zaira.materials.erratic_dynamic_attachment_expression,
179         pitch_handler__register_specifier__base_pitch="g'",
180     ),
181     piano_lh=new(
182         zaira.materials.piano_clusters_musicSpecifier,
183         attachment_handler__dynamic_expression=zaira.materials.erratic_dynamic_attachment_expression,
184         pitch_handler__register_specifier__base_pitch="c",
185         seed=1,
186     ),
187 )
188 )
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        fuse_groups=True,
        padding=durationtools.Duration(1, 4),
        reflect=True,
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(0, 2, 4),
        ratio=(1, 2, 1, 2, 1),
    ),
    piano_rh=zaira.materials.piano_guero_musicSpecifier,
    piano_lh=new(
        zaira.materials.piano_guero_musicSpecifier,
        seed=1,
    ),
)

### STRING SETTINGS ##################################################################

segment_maker.add_setting(
    timespan_maker=zaira.materials.granular_timespan_maker,
    timespan_identifier=timespanTools.Timespan(
        start_offset=fanfare_duration,
    ),
    violin=new(
        zaira.materials.string_trills_musicSpecifier,
        pitch_handler__registerSpecifier__base_pitch="c'",
        pitch_handler__pitchOperationSpecifier=pitchtools.PitchOperation(
            pitchtools.Inversion(),
        ),
        seed=0,
    ),
    viola=new(
        zaira.materials.string_trills_musicSpecifier,
        pitch_handler__registerSpecifier__base_pitch='c',
        pitch_handler__pitchOperationSpecifier=pitchtools.PitchOperation(
            pitchtools.Inversion(),
        ),
        seed=1,
    ),
    cello=new(
        zaira.materials.string_trills_musicSpecifier,
        pitch_handler__registerSpecifier__base_pitch='c',
        pitch_handler__pitchOperationSpecifier=pitchtools.PitchOperation(
            pitchtools.Inversion(),
        ),
        seed=2,
    ),
)
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sparse_timespan_maker,
        padding=durationtools.Duration(1, 4),
        reflect=True,
    ),
    timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
    violin=new(
        zaira.materials.string_flourish_music_specifier,
        pitch_handler__register_specifier__base_pitch="c'",
        pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
            pitchtools.Transposition(3),
        ),
        rhythm_maker__talea__denominator=16,
        seed=0,
    ),
    viola=new(
        zaira.materials.string_flourish_music_specifier,
        pitch_handler__register_specifier__base_pitch="c'",
        pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
            pitchtools.Transposition(3),
        ),
        rhythm_maker__talea__denominator=16,
        seed=1,
    ),
    cello=new(
        zaira.materials.string_flourish_music_specifier,
        pitch_handler__register_specifier__base_pitch='c',
        pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
            pitchtools.Transposition(3),
        ),
        rhythm_maker__talea__denominator=16,
        seed=2,
    ),
)

segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.tutti_timespan_maker,
        padding=durationtools.Duration(1, 4),
        playing_groupings=(1,),
        playing_talea__counts=(8,),
        repeat=False,
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3, 5),
        ratio=(1, 2, 1, 1, 1, 2, 1),
        mask_timespan=timespantools.Timespan(
            start_offset=fanfare_duration,
299  ),
300  ),
301  violin=new(
302       zaira.materials.string_tutti_overpressure_music_specifier,
303       pitch_handler__pitch_specifier='g',
304       seed=0,
305  ),
306  viola=new(
307       zaira.materials.string_tutti_overpressure_music_specifier,
308       pitch_handler__pitch_specifier='c',
309       seed=1,
310  ),
311  cello=new(
312       zaira.materials.string_tutti_overpressure_music_specifier,
313       pitch_handler__pitch_specifier='c,',
314       seed=2,
315  ),
316 )
317
318 ### SHAKER SETTINGS ####################################################################
319
320 segment_maker.add_setting(
321       timespan_maker=new(
322           zaira.materials.sparse_timespan_maker,
323           padding=durationtools.Duration(3, 8),
324       ),
325       timespan_identifier=timespantools.Timespan(
326           start_offset=fanfare_duration,
327       ),
328       clarinet=zaira.materials.brazil_nut_music_specifier,
329       flute=zaira.materials.brazil_nut_music_specifier,
330       violin=zaira.materials.brazil_nut_music_specifier,
331       viola=zaira.materials.brazil_nut_music_specifier,
332       woods=zaira.materials.wood_bamboo_music_specifier,
333 )
334
335 ### FANFARE SETTINGS ####################################################################
336
337 segment_maker.add_setting(
338       timespan_maker=consort.FloodedTimespanMaker(),
339       timespan_identifier=timespantools.Timespan(
340           stop_offset=fanfare_duration,
341       ),
342       piano_rh=new(
343           zaira.materials.piano_fanfare_music_specifier,
344           pitch_handler__registerSpecifier__base_pitch="g'",
345       ),
346       piano_lh=new(
347           zaira.materials.piano_fanfare_music_specifier,
348           pitch_handler__logical_tie_expressions=none,
```python
zaira.makers.Percussion.KICK_DRUM,
)
),
metals=new(

pirch_handler=pitch_handler__pitch_specifier=zaira.makers.Percussion.BRAKE_DRUM,
),
)}
}),
}
),
### DEPENDENT MUSIC SETTINGS ##################################################################

segment_maker.add_setting(

timespan_maker=zaira.makers.pedals_timespan_maker,

piano_pedals=zaira.makers.piano_pedals_music_specifier,
)

B.3.10 zaira.segments.segment_1

from abjad import new
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import mathtools
from abjad.tools import pitchtools
from abjad.tools import timespantools
import consort
import zaira

### SEGMENT MAKER ####################################################################

segment_maker = zaira.makers.ZairaSegmentMaker(

    tempo=indicatortools.Tempo((1, 4), 48),
)

ratio = mathtools.NonreducedRatio([1, 15])

segment_maker.desired_duration_in_seconds = (durationtools.Multiplier(sum(ratio), 91) *

    zaira.makers.total_duration_in_seconds
)

fanfare_duration = durationtools.Duration(1, 16)
```
### WINDS SETTINGS

```python
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.tutti_timespan_maker,
        playing_groupings=((1,),),
    ),
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3, 5),
        ratio=(1, 2, 1, 2, 1, 2, 1),
        mask_timespan=timespantools.Timespan(
            start_offset=fanfare_duration,
        ),
    ),
    clarinet=new(
        zaira.materials.wind_airtone_music_specifier,
        pitch_handler__registerSpecifier__base_pitch='D3',
    ),
    flute=zaira.materials.wind_airtone_music_specifier,
    oboe=new(
        zaira.materials.wind_airtone_music_specifier,
        pitch_handler__registerSpecifier__base_pitch='Bb3',
    )
)
```

### PERCUSSION SETTINGS

```python
segment_maker.add_setting(
    timespan_maker=new(
        zaira.materials.sustained_timespan_maker,
        fuse_groups=True,
        timespanSpecifier=consort.TimespanSpecifier(
            minimum_duration=durationtools.Duration(1, 4),
        ),
    ),
    metals=new(
        zaira.materials.percussion_superball_music_specifier,
        pitch_handler__pitch_specifier=zaira.makers.Percussion.TAM_TAM,
        seed=2,
    ),
)
```

```python
segment_maker.add_setting(
    timespan_maker=zaira.materials.sparse_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3),
        ratio=(1, 1, 1, 1, 1),
    ),
    metals=new(
        zaira.materials.percussion_brushed_music_specifier,
    ),
)```
pitch_handler__pitch_specifier=zaira.makers.Percussion.BASS_DRUM,

    seed=2,
)

segment_maker.add_setting(
    timespan_maker=zaira.materials.sparse_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1, 3),
        ratio=(2, 1, 2, 1, 1),
    ),
    drums=zaira.materials.drum_heartbeat_music_specifier,
)

### PIANO SETTINGS ##################################################################

segment_maker.add_setting(
    timespan_maker=new(zaira.materials.sustained_timespan_maker,
        fuse_groups=True,
        reflect=True,
    ),
    piano_rh=zaira.materials.piano_drone_music_specifier,
)

### STRING SETTINGS ##################################################################

segment_maker.add_setting(
    timespan_maker=new(zaira.materials.tutti_timespan_maker,
        padding=durationtools.Duration(1, 4),
        playing_groupings=(1,),
        reflect=True,
    ),
    timespan_identifier=timespantools.Timespan(
        start_offset=fanfare_duration,
    ),
    violin=new(zaira.materials.string_undergrowth_music_specifier,
        pitch_handler__registerSpecifier__base_pitch='g',
        seed=0,
    ),
    viola=new(zaira.materials.string_undergrowth_music_specifier,
        pitch_handler__registerSpecifier__base_pitch='c',
        seed=1,
    ),
    cello=new(zaira.materials.string_undergrowth_music_specifier,
pitch_handler__register_specifier__base_pitch='c',
seed=2,
)
)
segment_maker.add_setting(
timespan_maker=new(
zaira.materials.granular_timespan_maker,
playing_talea_counts=(2, 2, 3, 2, 7, 1, 3, 2, 1),
),
timespan_identifier=consort.RatioPartsExpression(
parts=(0, 2, 4),
ratio=(1, 1, 1, 1),
mask_timespan=timespantools.Timespan(
    start_offset=fanfare_duration,
),
),

cello=new(
zaira.materials.cello_solo_musicSpecifier,
pitch_handler__pitch_operation_specifier=pitchtools.PitchOperation(
    pitchtools.Transposition(3),
),
)
)
### SHAKER SETTINGS #################################################################

segment_maker.add_setting(
timespan_maker=new(
zaira.materials.tutti_timespan_maker,
playing_talea_counts=(3, 2, 3, 3, 2, 4),
playing_groupings=(1,),
padding=durationtools.Duration(3, 8),
),
clarinet=zaira.materials.brazil_nut_musicSpecifier,
flute=zaira.materials.brazil_nut_musicSpecifier,
violin=zaira.materials.brazil_nut_musicSpecifier,
viola=zaira.materials.brazil_nut_musicSpecifier,
woods=zaira.materials.wood_bamboo_musicSpecifier,
)
### FANFARE SETTINGS ##################################################################

segment_maker.add_setting(
timespan_maker=consort.FloodedTimespanMaker(),
timespan_identifier=timespantools.Timespan(
    stop_offset=fanfare_duration,
),
piano_rh=new(
zaira.materials.piano_fanfare_music_specifier,
pitch_handler__register_specifier__base_pitch="g'",
)
piano_lh=new(
  zaira.materials.piano_fanfare_music_specifier,
pitch_handler__logical_tie_expressions=
    zaira.materials.piano_fanfare_music_specifier
    .pitch_handler.logical_tie_expressions[::-1],
pitch_handler__pitch_specifier="g c a f d f e b e",
pitch_handler__register_specifier__base_pitch="g,,",
),
drums=new(
  zaira.materials.percussion_fanfare_music_specifier,
pitch_handler__pitch_specifier=zaira.makers.Percussion.KICK_DRUM,
),
metals=new(
  zaira.materials.percussion_fanfare_music_specifier,
pitch_handler__pitch_specifier=zaira.makers.Percussion.BRAKE_DRUM,
)

### DEPENDENT MUSIC SETTING ###################################################

B.3.11 zaira.segments.segment_j

# -*- encoding: utf-8 -*-
from abjad import new
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import mathtools
from abjad.tools import timespantools
import consort
import zaira

### SEGMENT MAKER ##################################################################

segment_maker = zaira.makers.ZairaSegmentMaker(
    permitted_time_signatures=(
        (2, 4),
        (3, 8),
        ),
    tempo=indicatortools.Tempo((1, 4), 84),
)

ratio = mathtools.NonreducedRatio([2])
segment_maker.desired_duration_in_seconds = ( 
    durationtools.Multiplier(sum(ratio), 91) * 
    zaira.materials.total_duration_in_seconds 
) 

fanfare_duration = durationtools.Duration(2, 16) 

### PERCUSSION SETTINGS ####################################################################

segment_maker.add_setting( 
    timespan_maker=new( 
        zaira.materials.sustained_timespan_maker, 
        fuse_groups=True, 
    ), 
    drums=new( 
        zaira.materials.percussion_superball_musicSpecifier, 
        pitch_handler__pitch_specifier=zaira.makers.Percussion.BASS_DRUM, 
    ), 
    metals=new( 
        zaira.materials.percussion_superball_musicSpecifier, 
        pitch_handler__pitch_specifier=zaira.makers.Percussion.TAM_TAM, 
    ) 
) 

### PIANO SETTINGS ######################################################################

segment_maker.add_setting( 
    timespan_maker=zaira.materials.sparse_timespan_maker, 
    piano_rh=new( 
        zaira.materials.piano_prepared_treble_musicSpecifier, 
        rhythm_maker=zaira.materials.sustained_rhythm_maker, 
    ), 
    piano_lh=new( 
        zaira.materials.piano_prepared_bass_musicSpecifier, 
        rhythm_maker=zaira.materials.sustained_rhythm_maker, 
    ) 
) 

### SHAKER SETTINGS ####################################################################

segment_maker.add_setting( 
    timespan_maker=consort.FloodedTimespanMaker(), 
    clarinet=new( 
        zaira.materials.brazil_nut_musicSpecifier, 
        rhythm_maker=zaira.materials.sustained_rhythm_maker, 
    ), 
    flute=new( 

zaira.materials.brazil_nut_music_specifier,
rhythm_maker=zaira.materials.sustained_rhythm_maker,
),
violin=new(
zaira.materials.brazil_nut_music_specifier,
rhythm_maker=zaira.materials.sustained_rhythm_maker,
),
viola=new(
zaira.materials.brazil_nut_music_specifier,
rhythm_maker=zaira.materials.sustained_rhythm_maker,
),

### FANFARE SETTINGS ####################################################################

segment_maker.add_setting(
  timespan_maker=consort.FloodedTimespanMaker(),
  timespan_identifier=timespantools.Timespan(
    stop_offset=fanfare_duration,
  ),
  piano_rh=new(
    zaira.materials.piano_fanfare_music_specifier,
    pitch_handler__register_specifier__base_pitch="g'",
  ),
  piano_lh=new(
    zaira.materials.piano_fanfare_music_specifier,
    pitch_handler__logical_tie_expressions=zaira.materials.piano_fanfare_music_specifier.
      pitch_handler.logical_tie_expressions[:=-1],
    pitch_handler__pitch_specifier="g c a f d f e b e",
    pitch_handler__register_specifier__base_pitch="g,,",
  ),
  drums=new(
    zaira.materials.percussion_fanfare_music_specifier,
    pitch_handler__pitch_specifier=zaira.makers.Percussion.KICK_DRUM,
  ),
  metals=new(
    zaira.materials.percussion_fanfare_music_specifier,
    pitch_handler__pitch_specifier=zaira.makers.Percussion.BRAKE_DRUM,
  )
)

### DEPENDENT MUSIC SETTINGS ####################################################################

segment_maker.add_setting(
  timespan_maker=zaira.materials.pedals_timespan_maker,
  piano_pedals=zaira.materials.piano_pedals_music_specifier,
)
```python
# -*- encoding: utf-8 -*-
from abjad import new
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import mathtools
timespantools
import consort
import zaira

### SEGMENT MAKER

segment_maker = zaira.makers.ZairaSegmentMaker(
    permitted_time_signatures=(
        (3, 8),
        (4, 8),
    ),
    tempo=indicatortools.Tempo((1, 4), 96),
)

ratio = mathtools.NonreducedRatio([3])

segment_maker.desired_duration_in_seconds = (  
    durationtools.Multiplier(sum(ratio), 91) *  
    zaira.materials.total_duration_in_seconds  
)

### FANFARE SETTINGS

segment_maker.add_setting(
    timespan_maker=consort.FloodedTimespanMaker(),
    timespan_identifier=timespantools.Timespan(
        stop_offset=durationtools.Duration(3, 8),
    ),
    piano_rh=new(
        zaira.materials.piano_fanfare_music_specifier,
        pitch_handler__register_specifier__base_pitch="g'",
        rhythm_maker__denominators=[8],
    ),
    piano_lh=new(
        zaira.materials.piano_fanfare_music_specifier,
        pitch_handler__logical_tie_expressions=  
        zaira.materials.piano_fanfare_music_specifier  
        .pitch_handler.logical_tie_expressions[:1],
        pitch_handler__pitch_specifier="g c a f d f e b e",
        pitch_handler__register_specifier__base_pitch="g,,",
        rhythm_maker__denominators=[8],
    ),
    drums=new(
```
zaira.materials.percussion_fanfare_music_specifier,
pitch_handler__pitch_specifier=zaira.makers.Percussion.KICK_DRUM,
rhythm_maker__denominators=[8],
},
metals=new(
  zaira.materials.percussion_fanfare_music_specifier,
pitch_handler__pitch_specifier=zaira.makers.Percussion.BRAKE_DRUM,
rhythm_maker__denominators=[8],
),
)

### PERSUSSION SETTINGS

segment_maker.add_setting(
  timespan_maker=consort.FloodedTimespanMaker(),
timespan_identifier=timespantools.Timespan(
    start_offset=durationtools.Duration(5, 16),
  ),
metals=new(
  zaira.materials.percussion_reiteration_music_specifier,
pitch_handler__pitch_specifier=zaira.makers.Percussion.BRAKE_DRUM,
),
)

### DEPENDENT MUSIC SETTINGS

segment_maker.add_setting(
  timespan_maker=zaira.materials.pedals_timespan_maker,
piano_pedals=zaira.materials.piano_pedals_music_specifier,
)

B.4 ZAIRA STYLESHEET SOURCE

B.4.1 STYLESHEET.ILY

#(define-markup-command (vstrut layout props)
 ()
 #:category other
"
@cindex creating vertical space in text

Create a box of the same height as the current font.”
(let ((ref-mrkp (interpret-markup layout props "fp")))
 (ly:make-stencil (ly:stencil-expr empty-stencil)
empty-interval
(ly:stencil-extent ref-mrkp Y))))
afterGraceFraction = #(cons 127 128)
#(set-global-staff-size 12)
\layout {
ragged-right = ##t

%% COMMON %%
\context {
    \Voice
    \consists Horizontal_bracket_ engraver
    \remove Forbid_line_break_ engraver
}

\context {
    \Staff
    \remove Time_signature_ engraver
}

\context {
    \Dynamics
    \remove Bar_ engraver
}

%% TIME SIGNATURE CONTEXT %%
\context {
    \name TimeSignatureContext
    \type Engraver_group
    \consists Axis_group_ engraver
    \consists Bar_number_ engraver
    \consists Mark_ engraver
    \consists Metronome_mark_ engraver
    \consists Time_signature_ engraver
    \override BarNumber.X-extent = #'(0 . 0)
    \override BarNumber.Y-extent = #'(0 . 0)
    \override BarNumber.extra-offset = #'(-8 . -4)
    \override BarNumber.font-name = "Didot Italic"
    \override BarNumber.font-size = 2
    \override BarNumber.stencil = #(make-stencil-circler 0.1 0.7 ly:text-interface::print)
    \override MetronomeMark.X-extent = #'(0 . 0)
    \override MetronomeMark.X-offset = 5
    \override MetronomeMark.Y-offset = -2.5
    \override MetronomeMark.break-align-symbols = #'(time-signature)
    \override MetronomeMark.font-size = 3
    \override RehearsalMark.X-extent = #'(0 . 0)
    \override RehearsalMark.Y-offset = 8
    \override RehearsalMark.break-align-symbols = #'(time-signature)
    \override RehearsalMark.break-visibility = #end-of-line-invisible
    \override RehearsalMark.font-name = "Didot"
    \override RehearsalMark.font-size = 10
    \override RehearsalMark.outside-staff-priority = 500
    \override RehearsalMark.self-alignment-X = #CENTER
\override TimeSignature.X-extent = #'(0 . 0)
\override TimeSignature.break-align-symbols = #'(staff-bar)
\override TimeSignature.break-visibility = #end-of-line-invisible
\override TimeSignature.font-size = 3
\override TimeSignature.style = #'numbered
\override VerticalAxisGroup.staff-staff-spacing = #'(
    (basic-distance . 8)
    (minimum-distance . 8)
    (padding . 8)
    (stretchability . 0)
)

%%% WINDS %%%

\context { \Staff \name FluteStaff \type Engraver_group \alias Staff }

\context { \Staff \name ClarinetInBFlatStaff \type Engraver_group \alias Staff }

\context { \Staff \name OboeStaff \type Engraver_group \alias Staff }

%%% DRUMS %%%

\context { \Staff \name MetalsStaff \type Engraver_group \alias Staff }

\context { \Staff \name WoodsStaff \type Engraver_group \alias Staff
\override BarLine.bar-extent = #'(-1 . 1)
\override StaffSymbol.line-count = 3
}
\context { \Staff 
  \name DrumsStaff 
  \type Engraver_group 
  \alias Staff 
  \override BarLine.bar-extent = #"(-1 . 1) 
  \override StaffSymbol.line-count = 3
}

%%% PIANO %%%
\context { 
  \Staff 
  \name PianoUpperStaff 
  \type Engraver_group 
  \alias Staff 
}
\context { 
  \Staff 
  \name PianoLowerStaff 
  \type Engraver_group 
  \alias Staff 
}
\context { 
  \PianoStaff 
  \name PianoPerformerGroup 
  \type Engraver_group 
  \alias PianoStaff 
  \accepts Staff 
  \accepts PianoUpperStaff 
  \accepts PianoLowerStaff 
  \override SystemStartBracket.stencil = ##f
}

%%% STRINGS %%%
\context { 
  \PianoStaff 
  \accepts PianoUpperStaff 
  \accepts PianoLowerStaff 
}

\context { 
  \Staff 
  \name StringStaff 
  \type Engraver_group 
  \alias Staff 
}
\context { 

defineStaff {Drums} {PianoUpper} {PianoLower} {Strings} 

\StaffGroup
\name StringPerformerGroup
\type Engraver_group
\alias StaffGroup
\accepts BowingStaff
\accepts FingeringStaff
\accepts StringStaff
}

%% ANNOTATIONS %%

\context {
\Voice
  \name InnerAnnotation
  \type Engraver_group
  \alias Voice
  \override Accidental.stencil = ##f
  \override Dots.stencil = ##f
  \override Flag.stencil = ##f
  \override NoteColumn.ignore-collision = ##t
  \override NoteHead.no-ledgers = ##t
  \override NoteHead.stencil = ##f
  \override Stem.stencil = ##f
  \override TupletBracket.dash-fraction = 0.125
  \override TupletBracket.dash-period = 1.0
  \override TupletBracket.outside-staff-padding = 1
  \override TupletBracket.outside-staff-priority = 999
  \override TupletBracket.style = #'dashed-line
  \override TupletNumber.stencil = ##f
}

\context {
\Voice
  \name OuterAnnotation
  \type Engraver_group
  \alias Voice
  \override Accidental.stencil = ##f
  \override Dots.stencil = ##f
  \override Flag.stencil = ##f
  \override NoteHead.no-ledgers = ##t
  \override NoteHead.stencil = ##f
  \override Stem.stencil = ##f
  \override TupletBracket.dash-fraction = 0.125
  \override TupletBracket.dash-period = 1.0
  \override TupletBracket.outside-staff-padding = 1
  \override TupletBracket.outside-staff-priority = 999
  \override NoteColumn.ignore-collision = ##t
}

\context {
\Staff
  \accepts InnerAnnotation
  \accepts OuterAnnotation
}

837
\context {
  \StaffGroup
  \name PerformerGroup
  \type Engraver_group
  \alias StaffGroup
  \accepts Staff
  \accepts FluteStaff
  \accepts OboeStaff
  \accepts ClarinetInBFlatStaff
  \accepts DrumsStaff
  \accepts WoodsStaff
  \accepts MetalsStaff
  \accepts StringStaff
}

\context {
  \StaffGroup
  \name EnsembleGroup
  \type Engraver_group
  \alias StaffGroup
  \accepts PerformerGroup
  \accepts StringPerformerGroup
  \accepts PianoPerformerGroup
}

\context {
  \Score
  \accepts TimeSignatureContext
  \accepts PerformerGroup
  \accepts EnsembleGroup
  \remove Metronome_mark_engraver
  \remove Mark_engraver
  \remove Bar_number_engraver
  \override BarLine.bar-extent = #'(-2 . 2)
  \override BarLine.hair-thickness = 0.5
  \override BarLine.space-alist = #'(time-signature extra-space . 0.0)
  \override Beam.beam-thickness = 0.75
  \override Beam.breakable = ##t
  \override Beam.length-fraction = 1.5
}
\override DynamicLineSpanner.add-stem-support = ##t
\override DynamicLineSpanner.outside-staff-padding = 1
\override Glissando.breakable = ##t
\override Glissando.thickness = 3
\override GraceSpacing.common-shortest-duration = #(ly:make-moment 1 16)
\override NoteCollision.merge-differently-dotted = ##t
\override NoteColumn.ignore-collision = ##t
\override OttavaBracket.add-stem-support = ##t
\override OttavaBracket.padding = 2
\override SpacingSpanner.base-shortest-duration = #(ly:make-moment 1 64)
\override SpacingSpanner.strict-grace-spacing = ##f
\override SpacingSpanner.strict-note-spacing = ##f
\override SpacingSpanner.uniform-stretching = ##t
\override Stem.details.beamed-lengths = #'(6)
\override Stem.details.lengths = #'(6)
\override Stem.stemlet-length = 1.5
\override StemTremolo.beam-width = 1.5
\override StemTremolo.flag-count = 4
\override StemTremolo.slope = 0.5
\override StemTremolo.style = '#default
\override SustainPedal.self-alignment-X = #CENTER
\override SustainPedalLineSpanner.padding = 2
\override SustainPedallineSpanner.to-barline = ##t
\override TrillPitchAccidental.avoid-slur = #'ignore
\override TrillPitchAccidental.layer = 1000
\override TrillPitchHead.layer = 1000
\override TrillPitchHead.whiteout = ##t
\override TrillSpanner.outside-staff-padding = 1
\override TrillSpanner.padding = 1
\override TupletBracket.avoid-scripts = ##t
\override TupletBracket.full-length-to-extent = ##t
\override TupletBracket.outside-staff-padding = 2
\override TupletBracket.padding = 2
\override TupletNumber.font-size = 1
\override VerticalAxisGroup.staff-staff-spacing = #'( (basic-distance . 8) (minimum-distance . 14) (padding . 4) (stretchability . 0) )
autoBeaming = ##f
pedalSustainStyle = #'mixed
proportionalNotationDuration = #(ly:make-moment 1 32)
tupletFullLength = ##t
}
}

839
\paper {
  \%\% MARGINS \%
  \% bottom-margin = 10\,mm
  \% left-margin = 30\,mm
  \% right-margin = 10\,mm
  \% top-margin = 10\,mm
  \%\% HEADERS AND FOOTERS \%
  evenFooterMarkup = \markup \fill-line {
    \concat {
      \bold \fontsize #3 \on-the-fly #print-page-number-check-first
      \fromproperty \#"page:page-number-string
      \hspace #18
    }
    " "
  }
  evenHeaderMarkup = \markup \fill-line {
    " "
  }
  oddFooterMarkup = \markup \fill-line {
    \concat {
      \bold \fontsize #3 \on-the-fly #print-page-number-check-first
      \fromproperty \#"page:page-number-string
      \hspace #18
    }
  }
  oddHeaderMarkup = \markup \fill-line {
    " "
  }
  print-first-page-number = ##f
  print-page-number = ##t
  \%\% PAGE BREAKING \%
  page-breaking = #ly:optimal-breaking
  ragged-bottom = ##f
  ragged-last-bottom = ##f
  \%\% SPACING DETAILS \%
  markup-system-spacing = #"{
    (basic-distance . 0)
    (minimum-distance . 12)
    (padding . 0)
    (stretchability . 0)
  }
  system-system-spacing = #"{
    (basic-distance . 8)
    (minimum-distance . 12)
    (padding . 4)
    (stretchability . 0)
}
top-markup-spacing = 
   (basic-distance . 0)
   (minimum-distance . 0)
   (padding . 8)
   (stretchability . 0)
)

top-system-spacing = 
   (basic-distance . 0)
   (minimum-distance . 10)
   (padding . 0)
   (stretchability . 0)
)

%%% ETC %%%

% system-separator-markup = \slashSeparator
This page intentionally left blank.
C

armilla source code

C.1 ARMILLA MAKERS SOURCE

C.1.1 armilla.makers.ArmillaScoreTemplate

```python
# -*- encoding: utf-8 -*-
from abjad import attach
from abjad import indicatortools
from abjad import instrumenttools
from abjad import markuptools
from abjad import scoretools
import consort

class ArmillaScoreTemplate(consort.ScoreTemplate):
    r'''A score template.
    ::
    >>> import armilla
    >>> template = armilla.makers.ArmillaScoreTemplate()
    >>> score = template()
    >>> print(format(score))
    \context Score = "Armilla Score" <<
    \tag #'time
    \context TimeSignatureContext = "Time Signature Context" {
    }\tag #'viola-1
    \context StringPerformerGroup = "Viola 1 Performer Group" \with {
        instrumentName = \markup {
            \hcenter-in
            #10
            "Viola 1"
    }
```
shortInstrumentName = \markup { \hcenter-in #10 "Va. 1"
}
} << \context BowingStaff = "Viola 1 Bowing Staff" {
\clef "percussion"
\context Voice = "Viola 1 Bowing Voice" {
}
}\context FingeringStaff = "Viola 1 Fingering Staff" {
\clef "alto"
\context Voice = "Viola 1 Fingering Voice" {
}
}

>>
\tag #'viola-2
\context StringPerformerGroup = "Viola 2 Performer Group" \with {
instrumentName = \markup { \hcenter-in #10 "Viola 2"
}
shortInstrumentName = \markup { \hcenter-in #10 "Va. 2"
}
} << \context BowingStaff = "Viola 2 Bowing Staff" {
\clef "percussion"
\context Voice = "Viola 2 Bowing Voice" {
}
}\context FingeringStaff = "Viola 2 Fingering Staff" {
\clef "alto"
\context Voice = "Viola 2 Fingering Voice" {
}
}

>>

::

>>> for item in sorted(template.context_name_abbreviations.items()):
... item
...
('viola_1', 'Viola 1 Performer Group')
('viola_1_lh', 'Viola 1 Fingering Voice')
('viola_1_rh', 'Viola 1 Bowing Voice')
('viola_2', 'Viola 2 Performer Group')
('viola_2_lh', 'Viola 2 Fingering Voice')
('viola_2_rh', 'Viola 2 Bowing Voice')

::

>>> for item in sorted(template.composite_context_pairs.items()):
...    item
... ('viola_1', ('viola_1_rh', 'viola_1_lh'))
('viola_2', ('viola_2_rh', 'viola_2_lh'))

'''

### CLASS VARIABLES ###

`__slots__ = ()`

### SPECIAL METHODS ###

```python
def __call__(self):
    manager = consort.ScoreTemplateManager
    time_signature_context = manager.make_time_signature_context()
    viola_one = manager.make_single_string_performer(
        clef=indicatortools.Clef('alto'),
        instrument=instrumenttools.Viola(
            instrument_name='viola 1',
            instrument_name_markup=markuptools.Markup('Viola 1').hcenter_in(10),
            short_instrument_name='va. 1',
            short_instrument_name_markup=markuptools.Markup('Va. 1').hcenter_in(10)
        ),
        split=True,
        score_template=self,
    )
    viola_two = manager.make_single_string_performer(
        clef=indicatortools.Clef('alto'),
        instrument=instrumenttools.Viola(
            instrument_name='viola 2',
            instrument_name_markup=markuptools.Markup('Viola 2').hcenter_in(10),
            short_instrument_name='va. 2',
            short_instrument_name_markup=markuptools.Markup('Va. 2').hcenter_in(10)
        ),
        split=True,
        score_template=self,
    )
    score = scoretools.Score([
C.1.2 armilla.makers.ArmillaSegmentMaker

```python
# -*- encoding: utf-8 -*-

from abjad import attach
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import markuptools
from abjad.tools import scoretools
import consort

class ArmillaSegmentMaker(consort.SegmentMaker):
    r'''Armilla segment maker.

    ::

        >>> import armilla
        >>> segment_maker = armilla.ArmillaSegmentMaker()
        >>> print(format(segment_maker))
        armilla.makers.ArmillaSegmentMaker(
            permitted_time_signatures=indicatortools.TimeSignatureInventory(
                [
                    indicatortools.TimeSignature((2, 4)),
                    indicatortools.TimeSignature((3, 4)),
                    indicatortools.TimeSignature((3, 8)),
                    indicatortools.TimeSignature((4, 8)),
                    indicatortools.TimeSignature((5, 8)),
                    indicatortools.TimeSignature((6, 8)),
                ]
            ),
            score_template=armilla.makers.ArmillaScoreTemplate(),
            timespan_quantization=durationtools.Duration(1, 8),
        )
```
```
### CLASS VARIABLES ###

```python
__slots__ = ('_repeat',)
```

### INITIALIZER ###

```python
def __init__(
    self,
    annotate_colors=None,
    annotate_phrasing=None,
    annotate_timespans=None,
    desired_duration_in_seconds=None,
    discard_final_silence=None,
    maximum_meter_run_length=None,
    name=None,
    omit_stylesheets=None,
    permitted_time_signatures=None,
    repeat=None,
    score_template=None,
    settings=None,
    tempo=None,
    timespan_quantization=None,
):
    import armilla
    permitted_time_signatures = (  
        permitted_time_signatures or  
        armilla.materials.time_signatures  
    )
    score_template = (  
        score_template or  
        armilla.makers.ArmillaScoreTemplate()  
    )
    timespan_quantization = (  
        timespan_quantization or  
        durationtools.Duration(1, 8),  
    )
    consort.SegmentMaker.__init__(
        self,
        annotate_colors=annotate_colors,
        annotate_phrasing=annotate_phrasing,
        annotate_timespans=annotate_timespans,
        desired_duration_in_seconds=desired_duration_in_seconds,
        discard_final_silence=discard_final_silence,
        maximum_meter_run_length=maximum_meter_run_length,
        name=name,
        omit_stylesheets=omit_stylesheets,
        permitted_time_signatures=permitted_time_signatures,
        repeat=repeat,
        score_template=score_template,
        settings=settings,
```
tempo=tempo,
timespan_quantization=timespan_quantization,
)
self.repeat = repeat

### PUBLIC METHODS ###

def postprocess_breath_marks(self, score):
    breath_mark = indicatortools.BreathMark()
    leaves = score['Viola 1 Bowing Voice'].select_leaves()
    if isinstance(leaves[-1], scoretools.Note):
        attach(breath_mark, leaves[-1], name='breath_mark')
    leaves = score['Viola 2 Bowing Voice'].select_leaves()
    if isinstance(leaves[-1], scoretools.Note):
        attach(breath_mark, leaves[-1], name='breath_mark')

def postprocess_left_hand_staff(self, staff):
    voice = staff[0]
    finger_pitches_voice = self.copy_voice(voice,
        attachment_names=('clef_spanner',
                          'staff_lines_spanner',
                          'trill_spanner',),
        new_voice_name=voice.name.replace('Fingering', 'LH Pitches'),
        new_context_name='FingeringPitchesVoice',
    )
    finger_spanner_voice = self.copy_voice(voice,
        attachment_names=('bend_after',
                          'glissando',),
        new_voice_name=voice.name.replace('Fingering', 'LH Spanner'),
        new_context_name='FingeringSpannerVoice',
        remove_grace_containers=True,
        remove_ties=True,
        replace_rests_with_skips=True,
    )
    voice_index = staff.index(voice)
    staff[voice_index:voice_index + 1] = [finger_pitches_voice,
                                          finger_spanner_voice]
    staff.is_simultaneous = True

def postprocess_right_hand_staff(self, staff):
    voice = staff[0]
    string_contact_voice = self.copy_voice(voice,
        attachment_names=('string_contact',),
    ),
new_voice_name = voice.name.replace('Bowing', 'RH String Contact'),
new_context_name = 'StringContactVoice',
remove_ties = True,
replace_rests_with_skips = True,
)
bow_contact_voice = self.copy_voice(
    voice,
    attachment_names=(
        'articulations',
        'bow_contact_spanner',
        'bow_contact_point',
        'bow_motion_technique',
        'breath_mark',
    ),
    new_voice_name = voice.name.replace('Bowing', 'RH Bow Contact'),
    new_context_name = 'BowContactVoice',
    remove_ties = True,
    replace_rests_with_skips = True,
)
bow_beaming_voice = self.copy_voice(
    voice,
    attachment_names=(
        'beam',
        'stem_tremolo',
    ),
    new_voice_name = voice.name.replace('Bowing', 'RH Beaming'),
    new_context_name = 'BowBeamingVoice',
    remove_ties = True,
)
bow_dynamics_voice = self.copy_voice(
    voice,
    attachment_names=(
        'dynamic_expression',
    ),
    new_voice_name = voice.name.replace('Bowing', 'RH Dynamics'),
    new_context_name = 'Dynamics',
    remove_ties = True,
    replace_rests_with_skips = True,
)
voice_index = staff.index(voice)
staff[voice_index:voice_index + 1] = [
    string_contact_voice,
    bow_contact_voice,
    bow_beaming_voice,
    bow_dynamics_voice,
]
staff.is_simultaneous = True

def configure_score(self):
    self.postprocess_breath_marks(self.score)
    self.postprocess_right_hand_staff(self.score['Viola 1 Bowing Staff'])
    self.postprocess_right_hand_staff(self.score['Viola 2 Bowing Staff'])
    self.postprocess_left_hand_staff(self.score['Viola 1 Fingering Staff'])
    self.postprocess_left_hand_staff(self.score['Viola 2 Fingering Staff'])
```python
consort.SegmentMaker.configure_score(self)

### PUBLIC PROPERTIES ###

@property
def final_markup(self):
    portland = markuptools.Markup('Portland, OR')
    queens = markuptools.Markup('Fresh Meadows, NY')
    date = markuptools.Markup('September 2014 - January 2015')
    null = markuptools.Markup.null()
    contents = [
        null,
        null,
        portland,
        queens,
        date,
    ]
    markup = markuptools.Markup.right_column(contents)
    markup = markup.italic()
    return markup

@property
def repeat(self):
    return self._repeat

@repeat.setter
def repeat(self, repeat):
    if repeat is not None:
        repeat = bool(repeat)
        self._repeat = repeat

@property
def score_package_name(self):
    return 'armilla'
```

C.2  **ARMILLA**

C.2.1  **ARMILLA.MATERIALS.DENSE_TIMESPAN MAKER**
playing_groupings=(3, 4, 2, 4),
silence_talea=rhythmmakertools.Talea(
    counts=(1, 1, 1, 2, 1, 1, 2),
    denominator=8,
),
)

C.2.2 ARMILLA.MATERIALS.INTERMITTENT_ACCENTS

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import selectortools

intermittent_accents = consort.AttachmentExpression(
    attachments=indicatortools.Articulation('>', 'down'),
    selector=selectortools.Selector().by_leaves()[:1].by_counts(
        [-3, 1, -2, 2, -1, 3, -4, 5],
    ).flatten(),
)

C.2.3 ARMILLA.MATERIALS.INTERMITTENT_CIRCULAR

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import selectortools

intermittent_circular = consort.AttachmentExpression(
    attachments=indicatortools.BowMotionTechnique('circular'),
    selector=selectortools.Selector().by_leaves()[:1].by_counts(
        [-3, 1, -4, 2, -1, 3, -5, 2],
    ).flatten(),
)

C.2.4 ARMILLA.MATERIALS.INTERMITTENT_GLISSANDI

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import selectortools
from abjad.tools import spannertools

intermittent_glissandi = consort.AttachmentExpression(
    attachments=spannertools.Glissando(
        allow_repeated_pitches=False,
        allow_ties=False,
        parenthesisize_repeated_pitches=True,
    ),
    selector=selectortools.Selector()\
        .by_leaves()\
        [:1]\
)
append_callback(consort.AfterGraceSelectorCallback())
.by_counts(
    [-3, 4, -2, 3, -1, 4],
cyclic=True,
.overhang=True,
.fuse_overhang=True,
.rotate=True,
),
)

C.2.5 armilla.materials.intermittent_tremoli

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import selectortools
from abjad.tools import spannertools

intermittent_tremoli = consort.AttachmentExpression(
    attachments=spannertools.StemTremoloSpanner(),
    selector=selectortools_SELECTOR().by_leaves()[::1].by_counts(
        [-7, 2, -3, 1],
    ),
)

C.2.6 armilla.materials.intermittent_trills

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import markuptools
from abjad.tools import schemetools
from abjad.tools import selectortools
from abjad.tools import spannertools

harmonic_fourth_trill = consort.ConsortTrillSpanner(
    interval='+P4',
    overrides={
        'trill_pitch_head__stencil': schemetools.Scheme(
            'ly:text-interface::print',
        ),
        'trill_pitch_head__text': markuptools.Markup.musicglyph(
            'noteheads.s@harmonic',
            direction=None,
        ),
    },
)
harmonic_third_trill = consort.ConsortTrillSpanner(
    interval='+m3',
    overrides={
        'trill_pitch_head__stencil': schemetools.Scheme(
            'ly:text-interface::print',
        ),
    },
)
C.2.7  armilla.materials.left_hand_diads_music_specifier

```python
'\texttt{trill_pitch_head__text}': markuptools.Markup.musicglyph(  
    'noteheads.s@harmonic',  
    direction=None,  
),  
)

stopped_third_trill = consort.ConsortTrillSpanner(  
    interval='m3',  
)

intermittent_trills = consort.AttachmentExpression(  
    attachments=(  
        harmonic_fourth_trill,  
        harmonic_third_trill,  
        harmonic_fourth_trill,  
        stopped_third_trill,  
        stopped_third_trill,  
    ),  
    selector=selectortools.Selector()  
        .by_leaves()  
        [:=-1]  
        .append_callback(consort.AfterGraceSelectorCallback())  
        .by_counts(  
            [-3, 2, -2, 1, -1, 2, -1, 1],  
            cyclic=True,  
            overhang=False,  
            fuse_overhang=False,  
            rotate=True,  
        ),  
),
```

```python
left_hand_diads_music_specifier = consort.MusicSpecifier(  
    attachment_handler=consort.AttachmentHandler(),  
    pitch_handler=consort.AbsolutePitchHandler(  
        deviations=(0, 0, 0, 0, 0.5, 0),  
        forbid_repetitions=True,  
        logical_tie_expressions=(  
            consort.ChordExpression(chord_expr=(0, 0)),  
            consort.ChordExpression(chord_expr=(0, 5)),  
            consort.ChordExpression(chord_expr=(0, 0)),  
            consort.ChordExpression(chord_expr=(0, 0)),  
            consort.ChordExpression(chord_expr=(0, 0)),  
            consort.ChordExpression(chord_expr=(0, 0)),
        ),
```
C.2.8  armilla.materials.left_hand_dietro_music_specifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools

left_hand_dietro_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        clef_spanner=consort.AttachmentExpression(
            clef=consort.ClefSpanner(
                'percussion',
                overrides={
                    'note_head__style': 'cross',
                },
            ),
            selector=selectortools.Selector().by_leaves(),
        ),
        pitch_handler=consort.AbsolutePitchHandler(
            logical_tie_expressions=(
                consort.ChordExpression("g b"),
            ))
    ),
    pitch_specifier=consort.PitchSpecifier(
        pitch_segments=(
            "a c' c' a a c' a g a c' c'",
            "a c' c' c' gs g c'",
            "a c' c' a a c' a g fs a a c' c'",
        ),
        ratio=(1, 1, 1),
    ),
    pitch_operation_specifier=consortPitchOperationSpecifier(
        pitch_operations=(
            None,
            pitchtools.PitchOperation(pitchtools.Rotation()),
            None,
            pitchtools.PitchOperation(pitchtools.Rotation()),
            None,
        ),
        ratio=(2, 1, 2, 2, 1),
    ),
    rhythm_maker=rhythmmakertools.NoteRhythmMaker(
        tie_specifier=rhythmmakertools.TieSpecifier(
            tie_across_divisions=False,
        ),
    ),
)
```
C.2.9 ARMILLA.MATERIALS.LEFT_HAND_GLISSANDI_MUSIC_SPECIFIER

```python
# -*- encoding: utf-8 -*-
import consortrom abjad.tools import durationtools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools

left_hand_glissandi_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        glissando=consort.AttachmentExpression(
            attachments=spannertools.Glissando(
                allow_repeated_pitches=False,
                allow_ties=False,
                parenthesize_repeated_pitches=True,
            ),
            selector=selectortools.Selector().by_leaves(
                ).append_callback(consort.AfterGraceSelectorCallback())
        ),
    ),
    grace_handler=consort.GraceHandler(
        counts=(0, 1, 2, 0, 0, 0),
        only_if_preceded_by_nonsilence=True,
    ),
    pitch_handler=consort.AbsolutePitchHandler(
        deviations=(0, 0, 0, 0, 0.5, 0),
        forbid_repetitions=True,
        grace_expressions=(
            consort.HarmonicExpression('P4'),
            consort.HarmonicExpression('M3'),
            consort.HarmonicExpression('P5'),
        ),
        pitch_segments=(
            "a c' a c''",
            "a g c' gs d'",
            "a c'",
        ),
        ratio=(1, 1, 1),
    ),
)
rhythm_maker=consort.CompositeRhythmMaker(
    last=rhythmmakertools.NoteRhythmMaker(),
    default=rhythmmakertools.EvenDivisionRhythmMaker(  
        denominators=(4,),
        extra_counts_per_division=(0, 0, 1, 0, 1, 0, 1, 0, 1),
        duration_spelling_specifier=rhythmmakertools.DurationSpellingSpecifier(  
            decrease_durations_monotonically=True,
            forbidden_written_duration=durationtools.Duration(1, 4),
        ),
    ),
),
)

C.2.10 armilla.materials.left_hand_pizzicati_music_specifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools

left_hand_pizzicati_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        arpeggio=indicatortools.Arpeggio(),
        clef_spanner=consort.AttachmentExpression(
            attachments=consort.ClefSpanner('treble'),
            selector=selectortools.Selector().by_leaves(),
        ),
    ),
    pitch_handler=consort.AbsolutePitchHandler(
        logical_tie_expressions=(
            consort.ChordExpression("cs' b' g'' ef'''"),
            consort.ChordExpression("fs e' c'' gs'''"),
            consort.ChordExpression("f ef' b' g''"),
            consort.ChordExpression("fs e' c'' gs'''"),
            consort.ChordExpression("c' bf' fs'' d'''"),
            consort.ChordExpression("cs' b' g'' ef'''"),
            consort.ChordExpression("f ef' b' g''"),
            consort.ChordExpression("fs e' c'' gs'''"),
            consort.ChordExpression("d' c'' gs'' e'''"),
        ),
    ),
    rhythm_maker=rhythmmakertools.IncisedRhythmMaker(
        incise_specifier=rhythmmakertools.InciseSpecifier(
            fill_with_notes=False,
            prefix_counts=(1, 1, 1, 1, 2, 1),
            prefix_talea=(1,),
            suffix_talea=(1,),
            suffix_counts=(0,),
            talea_denominator=16,
        ),
    ),
)
C.2.11  armilla.materials.left_hand_stasis_musicSpecifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import durationtools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools

left_hand_stasis_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        trill_spanner=consort.AttachmentExpression(
            attachments=(
                consort.ConsortTrillSpanner(interval='+m3'),
                consort.ConsortTrillSpanner(interval='+P4'),
                consort.ConsortTrillSpanner(interval='+m3'),
            ),
            selector=selectortools.Selector().by_leaves(),
        ),
    ),
    minimum_phrase_duration=durationtools.Duration(1, 4),
    pitch_handler=consort.AbsolutePitchHandler(
        deviations=(0, 0, 0.5),
        pitch_specifier="gs'",
    ),
    rhythm_maker=rhythmmakertools.NoteRhythmMaker(
        duration_spellingSpecifier=rhythmmakertools.DurationSpellingSpecifier(
            forbid_meter_rewriting=True,
        ),
    ),
)
```

C.2.12  armilla.materials.right_hand_circular_musicSpecifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad import durationtools
from abjad import indicatortools
from abjad import rhythmmakertools
from abjad import selectortools
from abjad import scoretools
from abjad import spannertools

right_hand_circular_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        bow_contact_points=consort.AttachmentExpression(
            attachments=(
                indicatortools.BowContactPoint(0),
                indicatortools.BowContactPoint(1),
                indicatortools.BowContactPoint((4, 5)),
                indicatortools.BowContactPoint((4, 5)),
                indicatortools.BowContactPoint((4, 5)),
                indicatortools.BowContactPoint((4, 5)),
            ),
        ),
    ),
)
```
indicatorools.BowContactPoint((4, 5)),
indicatorools.BowContactPoint((3, 5)),
indicatorools.BowContactPoint((4, 5)),
indicatorools.BowContactPoint((3, 5)),
indicatorools.BowContactPoint((3, 5)),
indicatorools.BowContactPoint((4, 5)),
indicatorools.BowContactPoint((3, 5)),
indicatorools.BowContactPoint((4, 5)),
),
selector=selectortools.Selector().by_leaves().flatten(),
bow_contact_spanner=spanntools.BowContactSpanner(),
bow_motion_techniques=consort.AttachmentExpression(
    attachments=indicatorools.BowMotionTechnique('circular'),
    selector=selectortools.Selector().by_leaves().flatten(),
),
dynamic_expressions=consort.DynamicExpression(
    dynamic_tokens='p ppp p ppp p ppp',
),
string_contact_points=consort.AttachmentExpression(
    attachments=(
        None,
        indicatorools.StringContactPoint('ordinario'),
        indicatorools.StringContactPoint('sul ponticello'),
        indicatorools.StringContactPoint('ordinario'),
        indicatorools.StringContactPoint('ordinario'),
        indicatorools.StringContactPoint('molto sul ponticello'),
    ),
    scope=scoretools.Voice,
    selector=selectortools.Selector().append_callback(
        consort.PhrasedSelectorCallback()).flatten(),
),
string_contact_spanner=consort.StringContactSpanner(),
rhythm_maker=consort.CompositeRhythmMaker(
    last=rhythmmakertools.IncisedRhythmMaker(
        incise_specifier=rhythmmakertools.InciseSpecifier(
            prefix_counts=[0],
            suffix_talea=[1],
            suffix_counts=[1],
            talea_denominator=16,
        ),
    ),
    default=rhythmmakertools.EvenDivisionRhythmMaker(
        denominators=(4,),
        extra_counts_per_division=(0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1),
        duration_spelling_specifier=rhythmmakertools.DurationSpellingSpecifier(
            decrease_durations_monotonically=True,
            forbidden_written_duration=durationtools.Duration(1, 4),
            forbid_meter_rewriting=True,
        ),
    ),
),
C.2.13  armilla.materials.right_hand_jete_music_specifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad import durationtools
from abjad import indicatortools
from abjad import rhythmmakertools
from abjad import scoretools
from abjad import selectortools
from abjad import spannertools

right_hand_jete_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        bow_contact_points=consort.AttachmentExpression(
            attachments=
                (indicatortools.BowContactPoint((4, 5)),
                indicatortools.BowContactPoint((3, 5)),
            ),
            selector=selectortools.Selector().by_leaves().flatten(),
        ),
        bow_contact_spanner=spannertools.BowContactSpanner(),
        bow_motion_techniques=consort.AttachmentExpression(
            attachments=indicatortools.BowMotionTechnique('jete'),
            selector=selectortools.Selector()  
                .by_leaves()
                .get_slice(stop=-1)
                .by_counts(
                    [2, 1, 2],
                    overhang=True,
                    fuse_overhang=True,
                    rotate=True,
                )
                .get_item(0, apply_to_each=False)
                .flatten(),
        ),
        dynamic_expressions=consort.AttachmentExpression(
            attachments=(
                indicatortools.Dynamic('mf'),
                indicatortools.Dynamic('mp'),
            ),
            selector=selectortools.Selector().by_leaves()[0]
        ),
        string_contact_points=consort.AttachmentExpression(
            attachments=(
                indicatortools.StringContactPoint('ordinario'),
                indicatortools.StringContactPoint('sul ponticello'),
                indicatortools.StringContactPoint('molto sul ponticello'),
                indicatortools.StringContactPoint('sul tasto'),
            ),
            scope=scoretools.Voice,
            selector=selectortools.Selector()
        ))
```

append_callback(consort.PhrasedSelectorCallback())

by_counts(
    [1, -1],
    cyclic=True,
    nonempty=True,
    overhang=True,
)

.flatten()

string_contact_spanner=consort.StringContactSpanner(),

rhythm_maker=consort.CompositeRhythmMaker(
    last=rhythmmakertools.IncisedRhythmMaker(
        inciseSpecifier=rhythmmakertools.InciseSpecifier(
            prefix_counts=[0],
            suffix_talea=[1],
            suffix_counts=[1],
            talea_denominator=16,
        ),
    ),
    default=rhythmmakertools.EvenDivisionRhythmMaker(
        denominators=(8, 8, 4),
        extra_counts_per_division=(0, 0, 0, 0, 0, 0),
        duration_spellingSpecifier=rhythmmakertools.DurationSpellingSpecifier(
            decrease_durations_monotonically=True,
            forbidden_written_duration=durationtools.Duration(1, 4),
            forbid_meter_rewriting=True,
        ),
    ),
)

C.2.14  | ARMILLA.MATERIALS.RIGHT_HAND_OVERPRESSURE_MUSIC_SPECIFIER

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools importscoretools
from abjad.tools importselectortools
from abjad.tools importspannertools

right_hand_overpressure_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        bow_contact_points=consort.AttachmentExpression(
            attachments=(
                indicatortools.BowContactPoint(0),
                indicatortools.BowContactPoint((1, 0)),
                indicatortools.BowContactPoint((1, 4)),
                indicatortools.BowContactPoint((3, 0)),
                indicatortools.BowContactPoint((1, 2)),
                indicatortools.BowContactPoint((5, 0)),
            ),
        ),
    ),
)
indicator/tools.BowContactPoint((3, 4)),
indicator/tools.BowContactPoint((7, 5)),
indicator/tools.BowContactPoint((1, 8)),
indicator/tools.BowContactPoint((0, 1)),
indicator/tools.BowContactPoint((3, 0)),
indicator/tools.BowContactPoint((4, 1)),
indicator/tools.BowContactPoint((1, 0)),
indicator/tools.BowContactPoint((3, 8)),
indicator/tools.BowContactPoint((1, 8)),
indicator/tools.BowContactPoint((0, 4)),
indicator/tools.BowContactPoint((3, 8)),
indicator/tools.BowContactPoint((1, 8)),
selector = selectortools.Selector().by_leaves().flatten(),
bow_contact_spanner = spannertools.BowContactSpanner(),
dynamic_expressions = consort.DynamicExpression(
    dynamic_tokens = 'p ppp p ppp f',
),
string_contact_points = consort.AttachmentExpression(
    attachments = {
        indicator/tools.StringContactPoint('ordinario'),
        indicator/tools.StringContactPoint('sul tasto'),
        indicator/tools.StringContactPoint('molto sul tasto'),
    },
    scope = scoretools.Voice,
),
selector = selectortools.Selector(
    consort.PhrasedSelectorCallback()
).by_counts(
    [1, -2, 1, -2, 1, -1],
    cyclic = True,
    nonempty = True,
    overhang = True,
).flatten(),
string_contact_spanner = consort.StringContactSpanner(),
rhythm_maker = consort.CompositeRhythmMaker(
    last = rhythmmakertools.IncisedRhythmMaker(
        inciseSpecifier = rhythmmakertools.InciseSpecifier(
            prefix_counts = [0],
            suffix_talea = [1],
            suffix_counts = [1],
            talea_denominator = 16,
        ),
    ),
    default = rhythmmakertools.EvenDivisionRhythmMaker(
        denominators = (4,),
    ),
extra_counts_per_division=(0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0),
duration_spellingSpecifier=rhythmmakertools.DurationSpellingSpecifier(
    decrease_durations_monotonically=True,
    forbidden_written_duration=durationtools.Duration(1, 4),
    forbid_meter_rewriting=True,
),
)
)

C.2.15  ARMILLA.MATERIALS.RIGHT_HAND_PIZZICATI_MUSIC_SPECIFIER

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import scoretools
from abjad.tools import selectortools
from abjad.tools import spannertools

right_hand_pizzicati_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        bow_contact_points=consort.AttachmentExpression(
            attachments=(indicatortools.BowContactPoint(None),),
            selector=selectortools.Selector().by_leaves().flatten(),
        ),
        bow_contact_spanner=spannertools.BowContactSpanner(),
        dynamic_expressions=consort.DynamicExpression(
            dynamic_tokens='mf',
        ),
        string_contact_points=consort.AttachmentExpression(
            attachments=indicatortools.StringContactPoint('pizzicato'),
            scope=scoretools.Voice,
            selector=selectortools.Selector(
                ).by_leaves()
            )[0]
        ),
        string_contact_spanner=consort.StringContactSpanner(),
    ),
    rhythm_maker=rhythmmakertools.IncisedRhythmMaker(
        inciseSpecifier=rhythmmakertools.InciseSpecifier(
            fill_with_notes=False,
            prefix_counts=(1, 1, 1, 1, 2, 1),
            prefix_talea=[1],
            suffix_talea=[1],
            suffix_counts=(0,),
            talea_denominator=16,
        ),
    ),
)
# -*- encoding: utf-8 -*-

import consort

from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import scoretools
from abjad.tools import selectortools
from abjad.tools import spannertools

right_hand_stasis_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        bow_contact_points=consort.AttachmentExpression(
            attachments=(
                indicatortools.BowContactPoint(1),
                indicatortools.BowContactPoint(1),
                indicatortools.BowContactPoint((4, 5)),
                indicatortools.BowContactPoint(1),
                indicatortools.BowContactPoint((4, 5)),
                indicatortools.BowContactPoint(1),
                indicatortools.BowContactPoint(1),
                indicatortools.BowContactPoint((2, 5)),
            ),
            selector=selectortools.Selector().by_leaves().flatten() ,
        ),
        bow_contact_spanner=spannertools.BowContactSpanner(),
        dynamic_expressions=consort.DynamicExpression(
            dynamic_tokens='p ppp',
        ),
        stem_tremolo_spanner=spannertools.StemTremoloSpanner(),
        string_contact_points=consort.AttachmentExpression(
            attachments=(
                indicatortools.StringContactPoint('ordinario'),
                indicatortools.StringContactPoint('sul tasto'),
                indicatortools.StringContactPoint('molto sul tasto'),
            ),
            scope=scoretools.Voice,
            selector=selectortools.Selector(
                ).append_callback(
                    consort.PhrasedSelectorCallback() )
            ).by_counts(
                [1, -2, 1, -2, 1, -1],
                cyclic=True,
                nonempty=True,
                overhang=True,
            ).flatten()
        ),
        string_contact_spanner=consort.StringContactSpanner(),
    ),
    minimum_phrase_duration=durationtools.Duration(1, 4),
    rhythm_maker=consort.CompositeRhythmMaker( 
    last=rhythmmakertools.IncisedRhythmMaker( 
inciseSpecifier=rhytmmakertools.InciseSpecifier(
    prefix_counts=[0],
    suffix_talea=[1],
    suffix_counts=[1],
    talea_denominator=16,
),
)

default=rhythmmakertools.EvenDivisionRhythmMaker(
    denominators={4,},
    extra_counts_per_division=(0, 0, 1, 0, 0, 1, 0, 1),
    duration_spellingSpecifier=rhythmmakertools.DurationSpellingSpecifier(
        decrease_durations_monotonically=True,
        forbidden_written_duration=durationtools.Duration(1, 4),
        forbid_meter_rewriting=True,
    ),
),
)

C.2.17 ARMILLA.MATERIALS.SPARSE_TIMESPAN_MAKER

sparse_timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=rhythmmakertools.Talea(
        counts=(1, 0),
        denominator=8,
    ),
    padding=durationtools.Duration(1, 8),
    playing_talea=rhythmmakertools.Talea(
        counts=(1, 1, 2),
        denominator=8,
    ),
    playing_groupings=(1, 2, 1, 2, 2, 3),
    silence_talea=rhythmmakertools.Talea(
        counts=(1, 1, 1, 2, 1, 1),
        denominator=8,
    ),
    timespanSpecifier=consort.TimespanSpecifier(
        minimum_duration=0,
    ),
)

C.2.18 ARMILLA.MATERIALS.SUSTAINED_TIMESPAN_MAKER

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import durationtools
from abjad.tools import rhytmmakertools

sparse_timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=rhythmmakertools.Talea(
        counts=(1, 0),
        denominator=8,
    ),
    padding=durationtools.Duration(1, 8),
    playing_talea=rhythmmakertools.Talea(
        counts=(1, 1, 2),
        denominator=8,
    ),
    playing_groupings=(1, 2, 1, 2, 2, 3),
    silence_talea=rhythmmakertools.Talea(
        counts=(1, 1, 1, 2, 1, 1),
        denominator=8,
    ),
    timespanSpecifier=consort.TimespanSpecifier(
        minimum_duration=0,
    ),
)
```python
sustained_timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=rhythmmakertools.Talea(
        counts=(5, 0),
        denominator=8,
    ),
    playing_talea=rhythmmakertools.Talea(
        counts=(5, 7, 4, 5),
        denominator=8,
    ),
    playing_groupings=(5, 6, 4, 8, 9),
    silence_talea=rhythmmakertools.Talea(
        counts=(3, 5, 3, 2, 5, 3),
        denominator=8,
    ),
    synchronize_step=True,
    timespanSpecifier=consort.TimespanSpecifier(
        minimum_duration=0,
    ),
)

C.2.19  ARMILLA.MATERIALS.SYNCHRONIZED_TIMESPAN MAKER

```
C.3 ARMILLA SEGMENTS SOURCE

C.3.1 ARMILLA SEGMENTS SEGMENT A_FAR_SORR

```python
# -*- encoding: utf-8 -*-
import armilla
import consort
from abjad import new
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import scoretools
from abjad.tools import selectortools

### SEGMENT MAKER ###

segment_maker = armilla.ArmillaSegmentMaker(
desired_duration_in_seconds=90,
discard_final_silence=True,
name='Far Sorr',
repeat=False,
tempo=indicatortools.Tempo((1, 4), 36),
)

### ATTACHMENTS ###

dietro_ponticello = consort.AttachmentExpression(
    attachments=indicatortools.StringContactPoint('dietro ponticello'),
    scope=scoretools.Voice,
    selector=selectortools.Selector().by_leaves(),
)

intermittent_accents = armilla.materials.intermittent_accents
intermittent_circular = armilla.materials.intermittent_circular
intermittent_glissandi = armilla.materials.intermittent_glissandi
intermittent_tremoli = armilla.materials.intermittent_tremoli

### MUSIC SPECIFIERS ###

lh_diads = new(
    armilla.materials.left_hand_diads_musicSpecifier,
    minimum_phrase_duration=durationtools.Duration((1, 4)),
)

lh_dietro = new(
    armilla.materials.left_hand_dietro_musicSpecifier,
    minimum_phrase_duration=durationtools.Duration((1, 4)),
)

lh_pizzicati = armilla.materials.left_hand_pizzicati_musicSpecifier
rh_overpressure = new(
    armilla.materials.right_hand_overpressure_musicSpecifier,
    minimum_phrase_duration=durationtools.Duration((1, 4)),
)
```

866
rh_pizzicati = armilla.materials.right_hand_pizzicati_musicSpecifier

### OVERPRESSURE ###

segment_maker.add_setting(
    timespan_maker=armilla.materials.sustained_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(0,),
        ratio=(1, 1, 1),
    ),
    viola_1=consort.CompositeMusicSpecifier(
        primary_musicSpecifier=new(
            rh_overpressure,
            attachment_handler__string_contact_points=dietro_ponticello,
        ),
        secondary_musicSpecifier=lh_dietro,
    ),
    viola_2=consort.CompositeMusicSpecifier(
        primary_musicSpecifier=armilla.materials.dense_timespan_maker,
        timespan_identifier=consort.RatioPartsExpression(
            parts=(1,),
            ratio=(1, 1, 1),
        ),
        viola_1=consort.CompositeMusicSpecifier(
            primary_musicSpecifier=new(
                rh_overpressure,
                attachment_handler__articulations=intermittent_accents,
                attachment_handler__stem_tremolo_spanner=intermittent_tremoli,
                rhythm_maker__default__denominators=(4, 4, 4, 8),
            ),
            rotation_indices=(1, 0, 1, 0, -1),
            secondary_musicSpecifier=new(
                lh_diads,
                attachment_handler__glissando=intermittent_glissandi,
            ),
        ),
        viola_2=consort.CompositeMusicSpecifier(
            primary_musicSpecifier=new(
                rh_overpressure,
                attachment_handler__articulations=intermittent_accents,
                attachment_handler__stem_tremolo_spanner=intermittent_tremoli,
                rhythm_maker__default__denominators=(4, 4, 4, 8),
            ),
            rotation_indices=(1, 0, 1, 0, -1),
            secondary_musicSpecifier=lh_diads,
        ),
    )
segment_maker.add_setting(
    timespan_maker=armilla.materials.sustained_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(2,),
        ratio=(1, 1, 1),
    ),
    viola_1=consort.CompositeMusicSpecifier(
        primary_music_specifier=new(
            rh_overpressure,
            attachment_handler__articulations=intermittent_accents,
            rhythm_maker__default__denominators=(4, 4, 4, 16, 4, 16),
        ),
        rotation_indices=(1, 0, 1, 0, -1),
        secondary_music_specifier=new(
            lh_diads,
            attachment_handler__glissando=intermittent_glissandi,
        ),
    ),
    viola_2=consort.CompositeMusicSpecifier(
        primary_music_specifier=new(
            rh_overpressure,
            attachment_handler__stem_tremolo_spanner=intermittent_tremoli,
            attachment_handler__bow_motion_technique_x=intermittent_circular,
        ),
        rotation_indices=(1, 0, 1, 0, -1),
        secondary_music_specifier=new(
            lh_diads,
            attachment_handler__glissando=intermittent_glissandi,
        ),
    ),
)

### PIZZICATI ###

segment_maker.add_setting(
    timespan_maker=armilla.materials.sparse_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(1,),
        ratio=(5, 1),
    ),
    viola_2=consort.CompositeMusicSpecifier(
        primary_music_specifier=rh_pizzicati,
        secondary_music_specifier=lh_pizzicati,
    ),
)

C.3.2 ARMILLA.SEVERS.SEGMENT_B_SELIDOR_A

```python
# -*- encoding: utf-8 -*-
import armilla
import consort
from abjad import new
from abjad.tools import indicatortools
```
segment_maker = armilla.ArmillaSegmentMaker(
    desired_duration_in_seconds=20 / 2,
    discard_final_silence=True,
    name='Selidor (i)',
    repeat=True,
    tempo=indicatortools.Tempo((1, 4), 72),
)

### ATTACHMENTS ###
intermittent_trills = armilla.materials.intermittent_trills

### MUSIC SPECIFIERS ###
rh_circular = armilla.materials.right_hand_circular_music_specifier
lh_glissandi = armilla.materials.left_hand_glissandi_music_specifier

### SETTINGS ###
segment_maker.add_setting(
    timespan_maker=armilla.materials.dense_timespan_maker,
    viola_1=consort.CompositeMusicSpecifier(
        primary_music_specifier=rh_circular,
        rotation_indices=(1, 0, 1, 0, -1),
        secondary_music_specifier=new(
            lh_glissandi,
            attachment_handler__trill_spanner=intermittent_trills,
        ),
    ),
    viola_2=consort.CompositeMusicSpecifier(
        primary_music_specifier=rh_circular,
        rotation_indices=(1, 0, 1, 0, -1),
        secondary_music_specifier=new(
            lh_glissandi,
            attachment_handler__trill_spanner=intermittent_trills,
        ),
    ),
)

C.3.3 armilla.segments.segment_c_wellogy
### SEGMENT MAKER ###

```python
segment_maker = armilla.ArmillaSegmentMaker(
    desired_duration_in_seconds=30 / 2,
    discard_final_silence=True,
    name='Wellogy',
    repeat=True,
    tempo=indicator_tools.Tempo((1, 4), 108),
)
```

### MUSIC SPECIFIERS ###

```python
lh_glissandi = new(
    armilla.materials.left_hand_glissandi_music_specifier,
    attachment_handler__bend_after=consort.AttachmentExpression,
       attachments=indicator_tools.BendAfter(4),
    selector=selector_tools.Selector().by_leaves([-1],
),
    pitch_handler__pitch_specifier="fs' gs' as'",
    rhythm_maker=rhythmmakertools.NoteRhythmMaker(
        duration_spelling_specifier=rhythmmakertools.DurationSpellingSpecifier(
            forbid_meter_rewriting=True,
        ),
    ),
)
```

### SETTINGS ###

```python
segment_maker.add_setting(
    timespan_maker=armilla.materials.sustained_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        ratio=(1, 2),
        parts=(1,),
    ),
    viola_1=consort.CompositeMusicSpecifier(
        discard_innerOffsets=True,
        primary_music_specifier=rh_stasis,
        secondary_music_specifier lh_stasis,
    ),
    viola_2=consort.CompositeMusicSpecifier(
        discard_innerOffsets=True,
        primary_music_specifier=rh_stasis,
        secondary_music_specifier lh_stasis,
    ),
)
```
timespan_identifier=consort.RatioPartsExpression(
    ratio=(1, 2),
    parts=(0,),
),
viola_1=consort.CompositeMusicSpecifier(
    primary_music_specifier=rh_jete,
    secondary_music_specifier=lh_glissandi,
),
viola_2=consort.CompositeMusicSpecifier(
    primary_music_specifier=rh_jete,
    secondary_music_specifier=lh_glissandi,
),
)

segment_maker.add_setting(
    timespan_maker=new(
        armilla.materials.sparse_timespan_maker,
        playing_talea_counts=(2, 1, 2, 1),
        playing_groupings=(1, 1, 2),
        silence_talea_denominator=2,
    ),
    timespan_identifier=consort.RatioPartsExpression(
        ratio=(2, 1),
        parts=(1,),
    ),
    viola_1=consort.CompositeMusicSpecifier(
        primary_music_specifier=rh_jete,
        secondary_music_specifier=lh_glissandi,
    ),
    viola_2=consort.CompositeMusicSpecifier(
        primary_music_specifier=rh_jete,
        secondary_music_specifier=lh_glissandi,
    ),
)

C.3.4  ARMILLA_SEGMENTS_SEGMENT_D_THE_LONG_DUNE_A

### SEGMENT MAKER ###
segment_maker = armilla.ArmillaSegmentMaker(
    desired_duration_in_seconds=120,
    discard_final_silence=True,
    name='The Long Dune (i)',
    repeat=False,
    tempo=indicatortools.Tempo((1, 4), 36),
)
### ATTACHMENTS ###

dietro_ponticello = consort.AttachmentExpression(
    attachments=indicatortools.StringContactPoint('dietro ponticello'),
    scope=scoretools.Voice,
    selector=indicatortools.Selector().by_leaves(),
)
dynamics_a = dynamic_expressions = consort.DynamicExpression(
    dynamic_tokens='p mf p ppp f p ff',
)
dynamics_b = dynamic_expressions = consort.DynamicExpression(
    dynamic_tokens='f p f mf ff p fff f fff f fff',
)
intermittent_accents = armilla.materials.intermittent_accents
intermittent_circular = armilla.materials.intermittent_circular
intermittent_tremoli = armilla.materials.intermittent_tremoli

### MUSIC SPECIFIERS ###

lh_diads = armilla.materials.left_hand_diads_musicSpecifier
lh_dietro = armilla.materials.left_hand_dietro_musicSpecifier
rh_overpressure = armilla.materials.right_hand_overpressure_musicSpecifier

### OVERPRESSURE ###

segment_maker.add_setting(
    timespan_maker=armilla.materials.sustained_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        parts=(0,),
        ratio=(3, 2, 1),
    ),
    viola_1=consort.CompositeMusicSpecifier(
        discard_inner_offsets=True,
        primary_musicSpecifier=new(
            rh_overpressure,
            attachment_handler__string_contact_points=dietro_ponticello,
        ),
        secondary_musicSpecifier lh_dietro,
    ),
    viola_2=consort.CompositeMusicSpecifier(
        discard_inner_offsets=True,
        primary_musicSpecifier=new(
            rh_overpressure,
            attachment_handler__string_contact_points=dietro_ponticello,
        ),
        secondary_musicSpecifier lh_dietro,
    ),
)
segment_maker.add_setting(
    timespan_maker=armilla.materials.dense_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
parts=(1,),
    ratio=(3, 2, 1),
),
    viola_1=consort.CompositeMusicSpecifier(
        discard_inner_offsets=True,
        primary_music_specifier=new(
            rh_overpressure,
            attachment_handler__articulations=intermittent_accents,
            attachment_handler__dynamic_expressions=dynamics_a,
            attachment_handler__string_contact_points=dietro_ponticello,
            rhythm_maker__default__denominators=(4, 4, 8, 4, 8),
        ),
        secondary_music_specifier=lh_dietsro,
    ),
    viola_2=consort.CompositeMusicSpecifier(
        primary_music_specifier=new(
            rh_overpressure,
            attachment_handler__bow_motion_technique_x=intermittent_circular,
            attachment_handler__dynamic_expressions=dynamics_a,
            attachment_handler__stem_tremolo_spanner=intermittent_tremoli,
            rhythm_maker__default__denominators=(4, 4, 8, 4, 8, 16),
        ),
        secondary_music_specifier=lh_diads,
    ),
    segment_maker.add_setting(
        timespan_maker=armilla.materials.sustained_timespan_maker,
        timespan_identifier=consort.RatioPartsExpression(
            parts=(2,),
            ratio=(3, 2, 1),
        ),
        viola_1=consort.CompositeMusicSpecifier(
            primary_music_specifier=new(
                rh_overpressure,
                attachment_handler__articulations=intermittent_accents,
                attachment_handler__bow_motion_technique_x=intermittent_circular,
                attachment_handler__dynamic_expressions=dynamics_b,
                attachment_handler__stem_tremolo_spanner=intermittent_tremoli,
                rhythm_maker__default__denominators=(4, 4, 16, 4, 16),
            ),
            rotation_indices=(1, 0, 1, 0, -1),
            secondary_music_specifier=lh_diads,
        ),
        viola_2=consort.CompositeMusicSpecifier(
            primary_music_specifier=new(
                rh_overpressure,
                attachment_handler__articulations=intermittent_accents,
                attachment_handler__bow_motion_technique_x=intermittent_circular,
                attachment_handler__dynamic_expressions=dynamics_b,
                attachment_handler__stem_tremolo_spanner=intermittent_tremoli,
                rhythm_maker__default__denominators=(4, 4, 16, 4, 16, 16),
            ),
    )
rotation_indices=(1, 0, 1, 0, -1),
secondary_music_specifier=lf_diads,
)

C.3.5 armilla.segments.segment_e_selidor_b

# -*- coding: utf-8 -*-
import armilla
import consort
from abjad import new
from abjad.tools import indicatortools

### SEGMENT MAKER ###

segment_maker = armilla.ArmillaSegmentMaker(
    desired_duration_in_seconds=40 / 2,
    discard_final_silence=True,
    name='Selidor (ii)',
    repeat=True,
    tempo=indicatortools.Tempo((1, 4), 72),
)

### ATTACHMENTS ###

intermittent_trills = armilla.materials.intermittent_trills

### MUSIC SPECIFIERS ###

lh_glissandi = armilla.materials.left_hand_glissandi_music_specifier
rh_circular = armilla.materials.right_hand_circular_music_specifier

### SETTINGS ###

segment_maker.add_setting(
    timespan_maker=armilla.materials.dense_timespan_maker,
    viola_1=consort.CompositeMusicSpecifier(
        primary_musicSpecifier=rh_circular,
        rotation_indices=(1, 0, 1, 0, -1),
        secondary_music_specifier=new(
            lh_glissandi,
            attachment_handler__trill_spanner=intermittent_trills,
        ),
    ),
    viola_2=consort.CompositeMusicSpecifier(
        primary_musicSpecifier=rh_circular,
        rotation_indices=(1, 0, 1, 0, -1),
        secondary_music_specifier=new(
            lh_glissandi,
            attachment_handler__trill_spanner=intermittent_trills,
        ),
    ),
)
```python
# -*- encoding: utf-8 -*-
import armilla
import consort
from abjad import new
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools

### SEGMENT MAKER ###

segment_maker = armilla.ArmillaSegmentMaker(
    desired_duration_in_seconds=40 / 2,
    discard_final_silence=True,
    name='The Isle of the Ear',
    repeat=True,
    tempo=indicatortools.Tempo((1, 4), 108),
)

### ATTACHMENTS ###

### MUSIC SPECIFIERS ###

lh_glissandi = new(
    armilla.materials.left_hand_glissandi_musicSpecifier,
    attachment_handler__bend_after=consort.AttachmentExpression(
        attachments=(
            indicatortools.BendAfter(4),
            indicatortools.BendAfter(-4),
        ),
        selector=selectortools.Selector().by_leaves()[[-1],
    ),
    pitch_handler__pitchSpecifier="fs' gs' as'",
)

rhythm_maker=rhythmmakertools.NoteRhythmMaker(
    duration_spellingSpecifier=rhythmmakertools.DurationSpellingSpecifier(  
        forbid_meter_rewriting=True,
    ),
)

lh_stasis = armilla.materials.left_hand_stasis_musicSpecifier
rh_jete = armilla.materials.right_hand_jete_musicSpecifier
rh_stasis = armilla.materials.right_hand_stasis_musicSpecifier

### SETTINGS ###

segment_maker.add_setting(
    timespan_maker=armilla.materials.sustained_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        ratio=(1, 2),
        parts=(1,),
    ),
)
```

\begin{verbatim}
viola_1 = consort.CompositeMusicSpecifier(
    discard_inner_offsets=True,
    primary_musicSpecifier=rh_stasis,
    rotation_indices=(1, 0, 1, 0, -1),
    secondary_musicSpecifier=lh_stasis,
),

viola_2 = consort.CompositeMusicSpecifier(
    discard_inner_offsets=True,
    primary_musicSpecifier=rh_stasis,
    rotation_indices=(1, 0, 1, 0, -1),
    secondary_musicSpecifier=lh_stasis,
),

segment_maker.add_setting(
    timespan_maker=armilla.materials.synchronized_timespan_maker,
    timespan_identifier=consort.RatioPartsExpression(
        ratio=(1, 3),
        parts=(0,)
    ),
    viola_1 = consort.CompositeMusicSpecifier(
        primary_musicSpecifier=rh_jete,
        secondary_musicSpecifier=lh_glissandi,
    ),
    viola_2 = consort.CompositeMusicSpecifier(
        primary_musicSpecifier=rh_jete,
        secondary_musicSpecifier=lh_glissandi,
    ),

segment_maker.add_setting(
    timespan_maker=new(
        armilla.materials.sparse_timespan_maker,
        padding=durationtools.Duration(1, 8),
        playing_talea_counts=(2, 1, 2, 1, 1),
        playing_groupings=(1, 1, 2),
        silence_talea_denominator=4,
    ),
    timespan_identifier=consort.RatioPartsExpression(
        ratio=(2, 1, 1),
        parts=(2,)
    ),
    viola_1 = consort.CompositeMusicSpecifier(
        primary_musicSpecifier=rh_jete,
        secondary_musicSpecifier=lh_glissandi,
    ),
    viola_2 = consort.CompositeMusicSpecifier(
        primary_musicSpecifier=rh_jete,
        secondary_musicSpecifier=lh_glissandi,
    ),
)
\end{verbatim}
```python
# -*- encoding: utf-8 -*-

import armilla
import consort
from abjad import new
from abjad.tools import indicatortools
from abjad.tools import spannertools
from abjad.tools import selectortools

### SEGMENT MAKER ###

segment_maker = armilla.ArmillaSegmentMaker(
    desired_duration_in_seconds=60 / 2,
    discard_final_silence=True,
    name='Selidor (iii)',
    repeat=True,
    tempo=indicatortools.Tempo((1, 4), 72),
)

### ATTACHMENTS ###

intermittent_trills = armilla.materials.intermittent_trills

### MUSIC SPECIFIERS ###

rh_circular = new(
    armilla.materials.right_hand_circular_musicSpecifier,
    attachment_handler__stem_tremolo_spanner=consort.AttachmentExpression(
        attachments=(
            None,
            spannertools.StemTremoloSpanner(),
        ),
        selector=selectortools.Selector().by_leaves().by_counts(
            [5, 1, 4, 2, 6, 3], cyclic=True,
        ),
    ),
)

lh_glissandi = new(
    armilla.materials.left_hand_glissandi_musicSpecifier,
    pitch_handler__pitchSpecifier=consort.PitchSpecifier(
        pitch_segments=(
            "a c' a a c' " ,
            "c' ef' g c' c' ef' " ,
            "a' c'' a' c'' c'' g' a' " ,
        ),
        ratio=(1, 1, 2),
    ),
    attachment_handler__trill_spanner=intermittent_trills,
)

### SETTINGS ###
```

877
### SEGMENT MAKER ###

segment_maker = armilla.ArmillaSegmentMaker(
    timespan_maker=armilla.materials.dense_timespan_maker,
    viola_1=consort.CompositeMusicSpecifier(
        primary_musicSpecifier=rh_circular,
        rotation_indices=(1, 0, 1, 0, -1),
        secondary_musicSpecifier=lh_glissandi,
    ),
    viola_2=consort.CompositeMusicSpecifier(
        primary_musicSpecifier=rh_circular,
        rotation_indices=(1, 0, 1, 0, -1),
        secondary_musicSpecifier=lh_glissandi,
    ),
)

### ATTACHMENTS ###

dietro_ponticello = consort.AttachmentExpression(
    attachments=indicatortools.StringContactPoint('dietro ponticello'),
    scope=scoretools.Voice,
    selector=selectortools.Selector().by_leaves(),
)

dynamics_a = dynamic_expressions = consort.DynamicExpression(
    dynamic_tokens='p mf p ppp f p ff',
)

dynamics_b = dynamic_expressions = consort.DynamicExpression(
    dynamic_tokens='f p f mf ff p fff f fff mf fff',
)

intermittent_accents = armilla.materials.intermittent_accents
intermittent_circular = armilla.materials.intermittent_circular
intermittent_glissandi = armilla.materials.intermittent_glissandi
intermittent_tremoli = armilla.materials.intermittent_tremoli
### MUSIC SPECIFIERS ###

lh_dietro = new(  
    armilla.materials.left_hand_dietro_musicSpecifier,  
    minimum_phrase_duration=durationtools.Duration(1, 4),  
)

lh_diads = new(  
    armilla.materials.left_hand_diads_musicSpecifier,  
    minimum_phrase_duration=durationtools.Duration(1, 4),  
)

rh_overpressure = new(  
    armilla.materials.right_hand_overpressure_musicSpecifier,  
    minimum_phrase_duration=durationtools.Duration(1, 4),  
)

rh_pizzicati = armilla.materials.right_hand_pizzicati_musicSpecifier
lh_pizzicati = armilla.materials.left_hand_pizzicati_musicSpecifier

### OVERPRESSURE ###

segment_maker.add_setting(  
    timespan_maker=armilla.materials.sustained_timespan_maker,  
    timespan_identifier=consort.RatioPartsExpression(  
        parts=[0,],  
        ratio=[2, 2, 1],  
    ),  
    viola_1=consort.CompositeMusicSpecifier(  
        discard_inner_offsets=True,  
        primary_music_specifier=new(  
            rh_overpressure,  
            attachment_handler__dynamic_expressions=dynamics_b,  
        ),  
        rotation_indices=[1, 0, 1, 0, -1],  
        secondary_music_specifier=lh_diads,  
    ),  
    viola_2=consort.CompositeMusicSpecifier(  
        discard_inner_offsets=True,  
        primary_music_specifier=new(  
            rh_overpressure,  
            attachment_handler__dynamic_expressions=dynamics_b,  
        ),  
        rotation_indices=[1, 0, 1, 0, -1],  
        secondary_music_specifier=lh_diads,  
    ),
)

segment_maker.add_setting(  
    timespan_maker=armilla.materials.dense_timespan_maker,  
    timespan_identifier=consort.RatioPartsExpression(  
        parts=[1,],  
        ratio=[2, 2, 1],  
    ),  
    viola_1=consort.CompositeMusicSpecifier(  
        primary_music_specifier=new(  
        ),
    )
rh_overpressure,
  attachment_handler__articulations=intermittent_accents,
  attachment_handler__dynamic_expressions=dynamics_a,
),
  rotation_indices=(1, 0, 1, 0, -1),
secondary_music_specifier=lh_diads,
),
viola_2=consort.CompositeMusicSpecifier(
  discard_inner_offsets=True,
  primary_music_specifier=new(
    rh_overpressure,
    attachment_handler__articulations=intermittent_accents,
    attachment_handler__dynamic_expressions=dynamics_a,
    seed=1,
  ),
  rotation_indices=(1, 0, 1, 0, -1),
  secondary_music_specifier=lh_diads,
),

segment_maker.add_setting(
  timespan_maker=armilla.materials.dense_timespan_maker,
  timespan_identifier=consort.RatioPartsExpression(
    parts=(2, ),
    ratio=(2, 2, 1),
  ),
viola_1=consort.CompositeMusicSpecifier(
  primary_music_specifier=new(
    rh_overpressure,
    attachment_handler__articulations=consort.AttachmentExpression(
      attachments=indicatortools.Articulation('>', 'down'),
      selector=selectortools_SELECTOR().by_leaves()[:].flatten(),
    ),
    attachment_handler__stem_tremolo_spanner=intermittent_tremoli,
    attachment_handler__bow_motion_technique_x=intermittent_circular,
    rhythm_maker__default__denominators=(4, 16, 4, 4, 4),
    seed=1,
  ),
  rotation_indices=(1, 0, 1, 0, -1),
  secondary_music_specifier=new(
    lh_diads,
    attachment_handler__glissando=intermittent_glissandi,
  ),
),
viola_2=consort.CompositeMusicSpecifier(
  primary_music_specifier=new(
    rh_overpressure,
    attachment_handler__articulations=consort.AttachmentExpression(
      attachments=indicatortools.Articulation('>', 'down'),
      selector=selectortools_SELECTOR().by_leaves()[:].flatten(),
    ),
    attachment_handler__stem_tremolo_spanner=intermittent_tremoli,
    attachment_handler__string_contact_points=dietro_ponticello,
    seed=2,
### PIZZICATI ###

''

C.4 ARMILLA

C.4.1 STYLESHEET.ily

```
#(define-markup-command (vstrut layout props)
  ()
#:category other
""
@cindex creating vertical space in text

7 Create a box of the same height as the current font."
(let ((ref (interpret-markup layout props "fp")))
  (ly:make-stencil (ly:stencil-expr empty-stencil)
    empty-interval
    (ly:stencil-extent ref mrkp Y))))

12 afterGraceFraction = #(cons 1023 1024)
#(set-default-paper-size "17x11" 'landscape)
#(set-global-staff-size 11.5)
\paper {
  bottom-margin = 10\mm
  left-margin = 10\mm
  right-margin = 10\mm
  top-margin = 10\mm
  evenFooterMarkup = \markup \fill-line {
    \concat {
      \bold \fontsize #3
      \on-the-fly #print-page-number-check-first
      \fromproperty #'page:page-number-string
    }
    ""
  }
  evenHeaderMarkup = \markup \fill-line { " " }
```

881
oddFooterMarkup = \markup \fill-line {
    "%" 
    \concat {
        \bold \fontsize #3
        \on-the-fly \print-page-number-check-first
        \fromproperty #'page:page-number-string
    }
}

oddHeaderMarkup = \markup \fill-line { "" }

print-first-page-number = ##f
print-page-number = ##t
page-breaking = #ly:optimal-breaking
ragged-bottom = ##f
ragged-last-bottom = ##t
markup-system-spacing = #'( (basic-distance . 0) (minimum-distance . 12) (padding . 0) (stretchability . 0) )
system-system-spacing = #'( (basic-distance . 12) (minimum-distance . 18) (padding . 12) (stretchability . 100) )


top-markup-spacing = #'( (basic-distance . 0) (minimum-distance . 0) (padding . 0) (stretchability . 0) )
top-system-spacing = #'( (basic-distance . 0) (minimum-distance . 10) (padding . 0) (stretchability . 0) )

\layout {
    \accidentalStyle forget
    \ragged-right = ##t

    \context {
        \name TimeSignatureContext
        \type Engraver_group
        \consists Axis_group_engraver
        \consists Bar_number_engraver
        \consists Mark_engraver
        \consists Metronome_mark_engraver
        \consists Script_engraver
    }

}
\consists Text_engraver
\consists Text_spanner_engraver
\consists Time_signature_engraver
\override BarNumber.Y-extent = #'(0 . 0)
\override BarNumber.Y-offset = 0
\override BarNumber.extra-offset = #'(-6 . 0)
\override BarNumber.font-name = "Didot Italic"
\override BarNumber.font-size = 1
\override BarNumber.padding = 4
\override BarNumber.stencil = #(make-stencil-circler 0.1 0.7 ly:text-interface::print)
\override MetronomeMark.X-extent = #'(0 . 0)
\override MetronomeMark.Y-extent = #'(0 . 0)
\override MetronomeMark.break-align-symbols = #'(left-edge)
\override MetronomeMark.extra-offset = #'(0 . 4)
\override MetronomeMark.font-size = 3
\override RehearsalMark.X-extent = #'(0 . 0)
\override RehearsalMark.X-offset = 6
\override RehearsalMark.Y-offset = -2.25
\override RehearsalMark.break-align-symbols = #'(time-signature)
\override RehearsalMark.break-visibility = #end-of-line-invisible
\override RehearsalMark.font-name = "Didot"
\override RehearsalMark.font-size = 8
\override RehearsalMark.outside-staff-priority = 500
\override RehearsalMark.self-alignment-X = #center
\override Script.extra-offset = #'(4 . -9)
\override Script.font-size = 6
\override TextScript.font-size = 3
\override TextScript.outside-staff-priority = 600
\override TextScript.padding = 6
\override TextSpanner.bound-details.right.attach-dir = #left
\override TextSpanner.padding = 6.75
\override TimeSignature.X-extent = #'(0 . 0)
\override TimeSignature.break-align-symbol = '#left-edge
\override TimeSignature.break-visibility = #end-of-line-invisible
\override TimeSignature.font-size = 3
\override TimeSignature.space-alist.clef = #'(extra-space . 0.5)
\override TimeSignature.style = #'numbered
\override VerticalAxisGroup.default-staff-staff-spacing = #'(basic-distance . 0)
  (minimum-distance . 14)
  (padding . 0)
  (stretchability . 0)
)
\override VerticalAxisGroup.minimum-Y-extent = #'(-20 . 20)
}

%%% DEFAULTS %%
\context {
  \Voice
    \remove Forbid_line_break_engraver
}
\context { 
  \Staff
\remove Time_signature_engraver
}

%% BOWING %%
\context {
  \Voice
  \name StringContactVoice
  \type Engraver_group
  \alias Voice
  \override Beam.stencil = ##f
  \override Dots.stencil = ##f
  \override Flag.stencil = ##f
  \override NoteHead.stencil = ##f
  \override Rest.stencil = ##f
  \override Stem.stencil = ##f
  \override TextScript.staff-padding = 7
  \override TextScript.self-alignment-X = #center
  \override TextSpanner.staff-padding = 7
  \override TextSpanner #'bound-details #'left #'attach-dir = 0
  \override TextSpanner #'bound-details #'right #'attach-dir = 0
  \override TupletBracket.stencil = ##f
  \override TupletNumber.stencil = ##f
}
\context {
  \Voice
  \name BowContactVoice
  \type Engraver_group
  \alias Voice
  \override Beam.stencil = ##f
  \override Dots.stencil = ##f
  \override Flag.stencil = ##f
  \override NoteHead.extra-offset = #'(0.05 . 0)
  \override Rest.stencil = ##f
  \override Script.staff-offset = 2.5
  \override Stem.stencil = ##f
  \override TupletBracket.stencil = ##f
  \override TupletNumber.stencil = ##f
}
\context {
  \Voice
  \name BowBeamingVoice
  \type Engraver_group
  \alias Voice
  \override Beam.direction = #down
  \override Beam.positions = #'(\-\-11 . \-\-11)
  \override Dots.X-offset = \-8
  \override Dots.staff-position = \-1
  \override Flag.Y-offset = \-10
  \override NoteHead.Y-offset = \-5
  \override NoteHead.stencil = ##f
  \override Stem.direction = #down
  \override Stem.length = 9
  \override Stem.stem-begin-position = \-11

\override TupletBracket.\positions = #'(-13 . -13)
\context {
\Dynamics
\remove Bar_\textit{engraver}
\override DynamicLineSpanner.\staff-padding = 11.5
\override DynamicText.\self-alignment-X = -1
\override Hairpin.\bound-padding = 1.5
%\override Hairpin.\minimum-length = 5
\override VerticalAxisGroup.\nonstaff-relatedstaff-\spacing = #'(
  (basic-distance . 5)
  (padding . 2.5)
)
\context {
\Staff
\name BowingStaff
\type Engraver_group
\alias Staff
\accepts Bow\textit{BeamingVoice}
\accepts Bow\textit{ContactVoice}
\accepts String\textit{ContactVoice}
\override Glissando.\bound-d\textit{etails}.left.padding = 0.75
\override Glissando.\bound-d\textit{etails}.right.padding = 0.75
\override Glissando.\thickness = 2
\override Glissando.\zigzag-length = 1.5
\override Glissando.\zigzag-width = 0.75
\override Parentheses\textit{Item}.\font-size = 1
\override Parentheses\textit{Item}.padding = 0.1
\override StaffSymbol.\textit{transparent} = ###t
}
%% FINGERING %%
\context {
\Voice
\name FingeringPitchesVoice
\type Engraver_group
\alias Voice
\override Beam.\direction = #down
\override Beam.\positions = #'(-9 . -9)
\override Stem.\direction = #down
% \override Tie.\textit{stencil} = ###f
\override Glissando.\textit{stencil} = ###f
\override TupletBracket.\positions = #'(-11 . -11)
}
\context {
\Voice
\name FingeringSpannerVoice
\type Engraver_group
\alias Voice
\override Beam.\direction = #down
\override Beam.\stencil = ###f
\override Dots.\textit{transparent} = ###t
}

\override Flag.stencil = ##f
\override NoteHead.transparent = ##t
\override Stem.direction = #down
\override Stem.stencil = ##f
\override Tie.stencil = ##f
\override TupletBracket.stencil = ##f
\override TupletNumber.stencil = ##f
}
\context {
\Staff
\name FingeringStaff
\type Engraver_group
\alias Staff
\accepts FingeringPitchesVoice
\accepts FingeringSpannerVoice
\override Glissando.bound-details.left.padding = 1.5
\override Glissando.bound-details.right.padding = 1.5
\override Glissando.thickness = 2
\override StaffSymbol.color = #(x11-color 'grey50)
}

%% STRING PERFORMER GROUP %%%

\context {
\StaffGroup
\name StringPerformerGroup
\type Engraver_group
\alias StaffGroup
\accepts BowingStaff
\accepts FingeringStaff
}

%% SCORE %%%

\context {
\Score
\accepts TimeSignatureContext
\accepts StringPerformerGroup
\remove Bar_number engraver
\remove Mark engraver
\remove Metronome_mark engraver
\override BarLine.bar-extent = #'(-2 . 2)
\override BarLine.hair-thickness = 0.5
\override BarLine.space-alist = #'(time-signature extra-space . 0.0)
(custos minimum-space . 0.0)
(clef minimum-space . 0.0)
(key-signature extra-space . 0.0)
(key-cancellation extra-space . 0.0)
(first-note fixed-space . 0.0)
(next-note semi-fixed-space . 0.0)
(right-edge extra-space . 0.0)
}
\override Beam.beam-thickness = 0.75

\context {
\Score
\accepts TimeSignatureContext
\accepts StringPerformerGroup
\remove Bar_number engraver
\remove Mark engraver
\remove Metronome_mark engraver
\override BarLine.bar-extent = #'(-2 . 2)
\override BarLine.hair-thickness = 0.5
\override BarLine.space-alist = #'(time-signature extra-space . 0.0)
(custos minimum-space . 0.0)
(clef minimum-space . 0.0)
(key-signature extra-space . 0.0)
(key-cancellation extra-space . 0.0)
(first-note fixed-space . 0.0)
(next-note semi-fixed-space . 0.0)
(right-edge extra-space . 0.0)
}
\context {
\Staff
\name FingeringStaff
\type Engraver_group
\alias Staff
\accepts FingeringPitchesVoice
\accepts FingeringSpannerVoice
\override Glissando.bound-details.left.padding = 1.5
\override Glissando.bound-details.right.padding = 1.5
\override Glissando.thickness = 2
\override StaffSymbol.color = #(x11-color 'grey50)
}

%% STRING PERFORMER GROUP %%%

\context {
\StaffGroup
\name StringPerformerGroup
\type Engraver_group
\alias StaffGroup
\accepts BowingStaff
\accepts FingeringStaff
}

%% SCORE %%%

\context {
\Score
\accepts TimeSignatureContext
\accepts StringPerformerGroup
\remove Bar_number engraver
\remove Mark engraver
\remove Metronome_mark engraver
\override BarLine.bar-extent = #'(-2 . 2)
\override BarLine.hair-thickness = 0.5
\override BarLine.space-alist = #'(time-signature extra-space . 0.0)
(custos minimum-space . 0.0)
(clef minimum-space . 0.0)
(key-signature extra-space . 0.0)
(key-cancellation extra-space . 0.0)
(first-note fixed-space . 0.0)
(next-note semi-fixed-space . 0.0)
(right-edge extra-space . 0.0)
}
\override Beam.beam-thickness = 0.75
\override Beam.breakable = ##t
\override Beam.length-fraction = 1.5
\override DynamicLineSpanner.add-stem-support = ##t
\override DynamicLineSpanner.outside-staff-padding = 1
\override Glissando.after-line-breaking = ##t
\override Glissando.breakable = ##t
\override Glissando.thickness = 3
\override GraceSpacing.common-shortest-duration = 1(ly:make-moment 1 8)
\override NoteCollision.merge-differently-dotted = ##t
\override NoteColumn.ignore-collision = ##t
\override OttavaBracket.add-stem-support = ##t
\override OttavaBracket.padding = 2
\override SpacingSpanner.base-shortest-duration = 1(ly:make-moment 1 64)
\override SpacingSpanner.strict-grace-spacing = ##f
\override SpacingSpanner.strict-note-spacing = ##f
\override SpacingSpanner.uniform-stretching = ##t
\override Stem.details.beamed-lengths = #'(6)
\override Stem.details.lengths = #'(6)
\override Stem.stemlet-length = 1.5
\override StemTremolo.beam-width = 1.5
\override StemTremolo.flag-count = 4
\override StemTremolo.slope = 0.5
\override StemTremolo.style = #'default
\override SustainPedal.self-alignment-X = CENTER
\override SustainPedallineSpanner.padding = 2
\override TextScript.add-stem-support = ##t
\override TextScript.outside-staff-padding = 1
\override TextScript.padding = 1
\override TextSpanner.staff-padding = 1
\override TextSpanner.bound-details.right.padding = 2.5
\override TrillPitchAccidental.avoid-slur = #'ignore
\override TrillPitchAccidental.layer = 1000
\override TrillPitchAccidental.whiteout = ##t
\override TrillPitchHead.layer = 1000
\override TrillPitchHead.whiteout = ##t
\override TrillSpanner.bound-details.right.padding = 1
\override TrillSpanner.outside-staff-padding = 1
\override TrillSpanner.padding = 1
\override TupletBracket.avoid-scripts = ##t
\override TupletBracket.direction = #down
\override TupletBracket.full-length-to-extent = ##t
\override TupletBracket.outside-staff-padding = 2
\override TupletBracket.padding = 2
\override TupletNumber.font-size = 1
\override TupletNumber.text = #tuplet-number::calc-fraction-text
\override VerticalAxisGroup.staff-staff-spacing = #'(0
  (minimum-distance . 8)
  (padding . 4)
  (stretchability . 0)
)
autoBeaming = ##f
doubleRepeatType = "\|.|:"
proportionalNotationDuration = #(ly:make-moment 1 16)
tupletFullLength = ##t
D.1 ersilia-makers

D.1.1 ersilia.makers.ErsiliaScoreTemplate

```python
# -*- encoding: utf-8 -*-

import consort

from abjad import attach
from abjad.tools import instrumenttools
from abjad.tools import markuptools
from abjad.tools import scoretools

class ErsiliaScoreTemplate(consort.ScoreTemplate):
    r'''Ersilia score template.
    
    ::
    >>> import ersilia
    >>> template = ersilia.makers.ErsiliaScoreTemplate()
    >>> score = template()
    >>> print(format(score))
    
    .. container:: context Score = "Ersilia Score"

    .. container:: tag #'time

    .. container:: context TimeSignatureContext = "Time Signature Context"

    .. container:: context WindSectionStaffGroup = "Wind Section Staff Group"

    .. container:: tag #'flute

    .. container:: context FluteStaff = "Flute Staff"

    .. container:: clef "treble"

    .. container:: set Staff.instrumentName = \markup { Flute }

    .. container:: set Staff.shortInstrumentName = \markup { Fl. }

    .. container:: context Voice = "Flute Voice"
```
```
\tag #'oboe
\context OboeStaff = "Oboe Staff" {
\clef "treble"
\set Staff.instrumentName = \markup { Oboe }
\set Staff.shortInstrumentName = \markup { Ob. }
\context Voice = "Oboe Voice" {
}
}\tag #'clarinet
\context ClarinetStaff = "Clarinet Staff" {
\clef "treble"
\set Staff.instrumentName = \markup {
\right-column
{ Bass
  Clarinet
}\}
\set Staff.shortInstrumentName = \markup { Bass cl. }
\context Voice = "Clarinet Voice" {
}
}\tag #'saxophone
\context SaxophoneStaff = "Saxophone Staff" {
\clef "treble"
\set Staff.instrumentName = \markup {
\right-column
{ Baritone
  Saxophone
}\}
\set Staff.shortInstrumentName = \markup { Bar. sax. }
\context Voice = "Saxophone Voice" {
}
}
}\context PercussionSectionStaffGroup = "Percussion Section Staff Group" <<
\tag #'guitar
\context GuitarStaffGroup = "Guitar Staff Group" <<
\context PitchPipes = "Guitar Pitch Pipe Staff" {
\clef "percussion"
\set Staff.instrumentName = \markup {
\right-column
{ Pitch
  Pipes
}\}
\set Staff.shortInstrumentName = \markup { Pp. }
\context Voice = "Guitar Pitch Pipe Voice" {
}
\context GuitarStaff = "Guitar Staff" {
  \clef "treble_8"
  \set Staff.instrumentName = \markup { Guitar }
  \set Staff.shortInstrumentName = \markup { Gt. }
  \context Voice = "Guitar Voice" {
  }
}
>
\tag #'piano
\context PianoStaffGroup = "Piano Staff Group" <<
\context PitchPipes = "Piano Pitch Pipe Staff" {
  \clef "percussion"
  \set Staff.instrumentName = \markup {
  \right-column
    { Pitch
      Pipes
    }
  }
  \set Staff.shortInstrumentName = \markup { Pp. }
  \context Voice = "Piano Pitch Pipe Voice" {
  }
}
\context PianoStaff = "Piano Staff" <<
  \set PianoStaff.instrumentName = \markup { Piano }
  \set PianoStaff.shortInstrumentName = \markup { Pf. }
  \context PianoUpperStaff = "Piano Upper Staff" {
    \clef "treble"
    \context Voice = "Piano Upper Voice" {
    }
  }
  \context PianoLowerStaff = "Piano Lower Staff" {
    \clef "bass"
    \context Voice = "Piano Lower Voice" {
    }
  }
  \context Dynamics = "Piano Pedals Voice" {
  }
>
\tag #'percussion
\context PercussionStaffGroup = "Percussion Staff Group" <<
\context PitchPipes = "Percussion Pitch Pipe Staff" {
  \clef "percussion"
  \set Staff.instrumentName = \markup {
  \right-column
    { Pitch
      Pipes
    }
  }
  \set Staff.shortInstrumentName = \markup { Pp. }
  \context Voice = "Percussion Pitch Pipe Voice" {
}
\context PercussionStaff = "Percussion Staff" {
\clef "percussion"
\set Staff.instrumentName = \markup { Percussion }
\set Staff.shortInstrumentName = \markup { Perc. }
\context Voice = "Percussion Voice" {
}
}

\context StringSectionStaffGroup = "String Section Staff Group" <<
\tag #'violin
\context ViolinStaff = "Violin Staff" {
\clef "treble"
\set Staff.instrumentName = \markup { Violin }
\set Staff.shortInstrumentName = \markup { Vn. }
\context Voice = "Violin Voice" {
}
}
\tag #'viola
\context ViolaStaff = "Viola Staff" {
\clef "alto"
\set Staff.instrumentName = \markup { Viola }
\set Staff.shortInstrumentName = \markup { Va. }
\context Voice = "Viola Voice" {
}
}
\tag #'cello
\context CelloStaff = "Cello Staff" {
\clef "bass"
\set Staff.instrumentName = \markup { Cello }
\set Staff.shortInstrumentName = \markup { Vc. }
\context Voice = "Cello Voice" {
}
}
\tag #'contrabass
\context ContrabassStaffGroup = "Contrabass Staff Group" <<
\context PitchPipes = "Contrabass Pitch Pipe Staff" {
\clef "percussion"
\set Staff.instrumentName = \markup {
\right-column
{
Pitch
Pipes
}
\set Staff.shortInstrumentName = \markup { Pp. }
\context Voice = "Contrabass Pitch Pipe Voice" {
}
}
\context ContrabassStaff = "Contrabass Staff" {
\clef "bass_8"
\set Staff.instrumentName = \markup { Contrabass }
}
\set Staff.shortInstrumentName = \markup { Cb. }
\context Voice = "Contrabass Voice" {
}
\>
\>
\>
\>
::

>>> for item in sorted(template.context_name_abbreviations.items()):
...    item

('bass', 'Contrabass Voice')
('bass_pp', 'Contrabass Pitch Pipe Voice')
('cello', 'Cello Voice')
('clarinet', 'Clarinet Voice')
('flute', 'Flute Voice')
('guitar', 'Guitar Voice')
('guitar_pp', 'Guitar Pitch Pipe Voice')
('oboe', 'Oboe Voice')
('percussion', 'Percussion Voice')
('percussion_pp', 'Percussion Pitch Pipe Voice')
('piano_lh', 'Piano Lower Voice')
('piano_pedals', 'Piano Pedals Voice')
('piano_pp', 'Piano Pitch Pipe Voice')
('piano_rh', 'Piano Upper Voice')
('saxophone', 'Saxophone Voice')
('viola', 'Viola Voice')
('violin', 'Violin Voice')

::

>>> for item in template.composite_context_pairs.items():
...    item

',

',

''

### CLASS VARIABLES ###

__slots__ = ()

### SPECIAL METHODS ###

def __call__(self):

    pitch_pipes = instrumenttools.Percussion(
        instrument_name='pitch pipes',
        instrument_name_markup=markuptools.Markup.right_column(
            ['Pitch', 'Pipes'],
            direction=None,
        ),
        short_instrument_name='pp',
    )
```
    time_signature_context = scoretools.Context(
        context_name="TimeSignatureContext",
        name="Time Signature Context",
    )
    self._attach_tag('time', time_signature_context)

    flute_staff = self._make_staff(
        'Flute', 'treble',
        instrument=instrumenttools.Flute(),
        tag='flute',
    )

    oboe_staff = self._make_staff(
        'Oboe', 'treble',
        instrument=instrumenttools.Oboe(),
        tag='oboe',
    )

    clarinet_staff = self._make_staff(
        'Clarinet', 'treble',
        instrument=instrumenttools.BassClarinet(
            instrument_name_markup=markuptools.Markup.right_column(
                ['Bass', 'Clarinet'],
                direction=None,
            ),
            tag='clarinet',
        )
    )

    saxophone_staff = self._make_staff(
        'Saxophone', 'treble',
        instrument=instrumenttools.BaritoneSaxophone(
            instrument_name_markup=markuptools.Markup.right_column(
                ['Baritone', 'Saxophone'],
                direction=None,
            ),
            tag='saxophone',
        )
    )

    wind_section_staff_group = scoretools.StaffGroup(
        [flute_staff,
         oboe_staff,
         clarinet_staff,
         saxophone_staff,
        ],
        context_name="WindSectionStaffGroup",
        name='Wind Section Staff Group',
    )

    guitar_staff = self._make_staff(
```
'Guitar', 'treble_8',
instrument=instrumenttools.Guitar(),
)
guitar_aux_staff = self._make_staff(
    'Guitar Pitch Pipe', 'percussion',
    abbreviation='guitar_pp',
    context_name='Pitch Pipes',
    instrument=pitch_pipes,
)
guitar_staff_group = scoretools.StaffGroup(
    [guitar_aux_staff, guitar_staff],
    name='Guitar Staff Group',
    context_name='GuitarStaffGroup',
)
sel._attach_tag('guitar', guitar_staff_group)
piano_staff = scoretools.StaffGroup(
    [piano_rh_staff, piano_lh_staff, piano_pedals],
    context_name='Piano Staff',
)
attach(instrumenttools.Piano(), piano_staff)
piano_staff_group = scoretools.StaffGroup(
    [piano_aux_staff, piano_staff],
    context_name='PianoStaffGroup',
    name='Piano Staff Group',
)
sel._attach_tag('piano', piano_staff_group)
percussion_aux_staff = self._make_staff(
    'Percussion Pitch Pipe', 'percussion',
    abbreviation='percussion_pp',
)
context_name='Pitch Pipes',
instrument=pitch_pipes,
)
percussion_staff_group = scoretools.StaffGroup(
    [percussion_aux_staff, percussion_staff],
    name='Percussion Staff Group',
    context_name='PercussionStaffGroup',
)
self._attach_tag('percussion', percussion_staff_group)

percussion_section_staff_group = scoretools.StaffGroup(
    [guitar_staff_group, piano_staff_group, percussion_staff_group],
    context_name='PercussionSectionStaffGroup',
    name='Percussion Section Staff Group',
)

violin_staff = self._make_staff(
    'Violin', 'treble',
    instrument=instrumenttools.Violin(),
    tag='violin',
)

viola_staff = self._make_staff(
    'Viola', 'alto',
    instrument=instrumenttools.Viola(),
    tag='viola',
)

cello_staff = self._make_staff(
    'Cello', 'bass',
    instrument=instrumenttools.Cello(),
    tag='cello',
)

contrabass_aux_staff = self._make_staff(
    'Contrabass Pitch Pipe',
    'percussion',
    abbreviation='bass_pp',
    context_name='Pitch Pipes',
    instrument=pitch_pipes,
)

contrabass_staff = self._make_staff(
    'Contrabass', 'bass_8',
    abbreviation='bass',
    instrument=instrumenttools.Contrabass(  
        pitch_range=['E1', 'G4'],
    ),
)

contrabass_staff_group = scoretools.StaffGroup(
    [contrabass_aux_staff, contrabass_staff],
)
        name='Contrabass Staff Group',
        context_name='ContrabassStaffGroup',
    )
    self._attach_tag('contrabass', contrabass_staff_group)

    string_section_staff_group = scoretools.StaffGroup(
        [violin_staff,
         viola_staff,
         cello_staff,
         contrabass_staff_group,
        ],
        context_name='StringSectionStaffGroup',
        name='String Section Staff Group',
    )

    score = scoretools.Score(
        [time_signature_context,
         wind_section_staff_group,
         percussion_section_staff_group,
         string_section_staff_group,
        ],
        name='Ersilia Score',
    )

    return score

D.1.2 ersilia.makers.ErsiliaSegmentMaker

# -*- encoding: utf-8 -*-
import collections
from abjad import iterate
from abjad.tools import durationtools
from abjad.tools import markuptools
from abjad.tools import scoretools
from abjad.tools import systemtools
import consort

class ErsiliaSegmentMaker(consort.SegmentMaker):
    ### CLASS VARIABLES ###
    __slots__ = ()
    ### INITIALIZER ###
    def __init__(
        self,
        annotate_colors=None,
        annotate_phrasing=None,
        annotate_timespans=None,
        desired_duration_in_seconds=None,
import ersilia

score_template = score_template or ersilia.ErsiliaScoreTemplate()

consort.SegmentMaker.__init__(
    self,
    annotate_colors=annotate_colors,
    annotate_phrasing=annotate_phrasing,
    annotate_timespans=annotate_timespans,
    desired_duration_in_seconds=desired_duration_in_seconds,
    discard_final_silence=discard_final_silence,
    maximum_meter_run_length=maximum_meter_run_length,
    name=name,
    permitted_time_signatures=permitted_time_signatures,
    repeat=repeat,
    score_template=score_template,
    settings=settings,
    tempo=tempo,
    timespan_quantization=durationtools.Duration(1, 8),
)

### PUBLIC METHODS ###

@staticmethod
def validate_score(score, verbose=True):
    consort.SegmentMaker.validate_score(score, verbose=verbose)

component = score['Piano Staff']

progress_indicator = systemtools.ProgressIndicator(
    is_warning=True,
    message=' coloring piano conflicts',
    verbose=verbose,
)

with progress_indicator:
    for vertical_moment in iterate(component).by_vertical_moment():
        pitch_numbers = collections.Counter()
        notes_and_chords = vertical_moment.notes_and_chords
        for note_or_chord in notes_and_chords:
            if isinstance(note_or_chord, scoretools.Note):
                pitch_number = note_or_chord.written_pitch.pitch_number
                pitch_number = float(pitch_number)
                pitch_numbers[pitch_number] += 1
            else:
                for pitch in note_or_chord.written_pitches:
                    pitch_number = pitch.pitch_number
                    pitch_number = float(pitch_number)
                    pitch_numbers[pitch_number] += 1

        conflict_pitch_numbers = set()
for pitch_number, count in pitch_numbers.items():
    if 1 < count:
        conflict_pitch_numbers.add(pitch_number)
    if not conflict_pitch_numbers:
        continue
for note_or_chord in notes_and_chords:
    if isinstance(note_or_chord, scoretools.Note):
        pitch_number = note_or_chord.written_pitch.pitch_number
        pitch_number = float(pitch_number)
        if pitch_number in conflict_pitch_numbers:
            #note_or_chord.note_head.tweak.color = 'red'
            progress_indicator.advance()
    else:
        for note_head in note_or_chord.note_heads:
            pitch_number = note_head.written_pitch.pitch_number
            pitch_number = float(pitch_number)
            if pitch_number in conflict_pitch_numbers:
                #note_head.tweak.color = 'red'
                progress_indicator.advance()

### PUBLIC PROPERTIES ###

@property
def final_markup(self):
    portland = markuptools.Markup('Portland, OR')
    date = markuptools.Markup('January 2015 - April 2015')
    null = markuptools.Markup.null()
    contents = [
        null, null, null, portland, date,
    ]
    markup = markuptools.Markup.right_column(contents)
    markup = markup.italic()
    return markup

@property
def score_package_name(self):
    return 'ersilia'

D.1.3 ersilia.makers.Percussion

# -*- encoding: utf-8 -*-
from abjad.tools import abctools
from abjad.tools import pitchtools

class Percussion(abctools.AbjadObject):
    WOOD_BLOCK_5 = pitchtools.NamedPitch('G4')
    WOOD_BLOCK_4 = pitchtools.NamedPitch('E4')
    WOOD_BLOCK_3 = pitchtools.NamedPitch('C4')
WOOD_BLOCK_2 = pitchtools.NamedPitch('A3')
WOOD_BLOCK_1 = pitchtools.NamedPitch('F3')
TOM_4 = pitchtools.NamedPitch('F4')
TOM_3 = pitchtools.NamedPitch('D4')
TOM_2 = pitchtools.NamedPitch('B3')
TOM_1 = pitchtools.NamedPitch('G3')
BAMBOO_WIND_CHIMES = pitchtools.NamedPitch('F4')
SNARE_DRUM = pitchtools.NamedPitch('D4')
TAM_TAM = pitchtools.NamedPitch('B3')
BASS_DRUM = pitchtools.NamedPitch('G3')

D.2 **ERSILIA** MATERIALS SOURCE

D.2.1 **ERSILIA.MATERIALS.ABBREVIATIONS**

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import markuptools
from abjad.tools import pitchtools
from abjad.tools import selectortools
from abjad.tools import spannertools

laissez_vibrer = consort.AttachmentExpression(
    attachments=[
        indicatortools.LaissezVibrer(),
        markuptools.Markup('L.V', Up)
            .caps()
            .tiny()
            .pad_around(0.5)
            .box()
            .pad_around(0.5)
    ],
    selector=selectortools.Selector()
        .by_logical_tie(pitched=True)
        .by_contiguity()
        .by_length('==', 1)
        .by_leaves()
        [0]
)
black_keys_spanner = spannertools.make_solid_text_spanner_with_nib(
    markuptools.Markup.flat().vcenter(),
)
chromatic_keys_spanner = spannertools.make_solid_text_spanner_with_nib(
    markuptools.Markup.concat([
        markuptools.Markup.natural(),
    ]),
)
```
markuptools.Markup.hspace(0.1),
markuptools.Markup.flat(),
]).vcenter(),
)

white_keys_spanner = spannertools.make_solid_text_spanner_with_nib(
    markuptools.Markup.natural().vcenter(),
)

percussion_staff = consort.AttachmentExpression(
    attachments=[
        spannertools.StaffLinesSpanner(
            lines=[-4, 4],
            overrides={
                'note_head__style': 'cross',
                'note_head__no_ledgers': True,
            },
        ),
        consort.ClefSpanner('percussion'),
    ],
)

agitato_pitch_specifier = consort.PitchSpecifier(
    pitch_segments=[
        pitchtools.PitchClassSegment([0, 3, 2, 5, 11, 1]),
        pitchtools.PitchClassSegment([11, 9]),
        pitchtools.PitchClassSegment([2, 4, 5, 8]),
        pitchtools.PitchClassSegment([0, 3, 5]),
        pitchtools.PitchClassSegment([2, 4, 5, 8]),
    ],
    ratio=[1, 2, 1, 2, 1],
)

pitch_operation_specifier = consort.PitchOperationSpecifier(
    pitch_operations=[
        pitchtools.Rotation(1),
        None,
        pitchtools.PitchOperation([
            pitchtools.Transposition(1),
            pitchtools.Inversion(),
        ]),
        None,
        pitchtools.Rotation(-1),
        pitchtools.Retrogression(),
    ],
    ratio=(1, 3, 1, 1, 2, 1),
)

def make_text_markup(text):
    markup = markuptools.Markup.concat([
        markuptools.Markup(r'\vstrut'),
        markuptools.Markup(text),
    ])
    markup = markup.smaller().italic().pad_around(0.5).whiteout().box()
    markup = markuptools.Markup(markup, Up)
    return markup

def make_text_spanner(text):
    markup_contents = make_text_markup(text).contents
    markup = markuptools.Markup(markup_contents)
    text_spanner = consort.ComplexTextSpanner(markup=markup)
    return text_spanner

guitar_chords = tuple(
pitchtools.PitchSegment(_) for _ in (  
    "d c' f' a'"",
    "df bf e' a'"",
    "g b f' e' a'"",
    "b g a d' af'"",
    "g b e' a'"",
    "f bf ef' g' c'"",
    "e a d' fs' b'"",
    "ef af df' f' bf'"",
    "d g c' e' a'"",
    "d b d' f' a'"
    "d f c' d' g'"
)
)

__all__ = [
    'agitato_pitch_specifier',
    'black_keys_spanner',
    'chromatic_keys_spanner',
    'guitar_chords',
    'laissez_vibrer',
    'make_text_markup',
    'make_text_spanner',
    'percussion_staff',
    'pitch_operation_specifier',
    'white_keys_spanner',
]

D.2.2 ersilia.materials.dense_timespan_maker

# -*- encoding: utf-8 -*-
import consort
from abjad import *


D.2.3 ersilia.materials.guitar_agitato_musicSpecifier

```python
# -*- encoding: utf-8 -*-
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import scoretools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations
import consort

guitar_agitato_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('==', (1, 8), preprolated=True)
            [0],
        ),
        dynamic_expressions=consort.DynamicExpression(
            dynamic_tokens='mf mp fff',
            start_dynamic_tokens='f',
            stop_dynamic_tokens='mf',
        ),
        mordent=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('mordent'),
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('>=', (1, 8), preprolated=True)
        ),
    ),
    playing_talea=rhythmmakertools.Talea(
        counts=(3, 4, 2, 2, 3, 3, 2),
        denominator=8,
    ),
    playing_groupings=(2, 1, 2, 3, 1, 1, 2, 2),
    repeat=True,
    silence_talea=rhythmmakertools.Talea(
        counts=(1, 2, 3, 1, 2, 5),
        denominator=8,
    ),
)```
by_class(scoretools.Note)
[0],

snappizzicato=consort.AttachmentExpression(
    attachments=indicatortools.Articulation('snappizzicato'),
    selector=selectortools.Selector()
    .by_logical_tie(pitched=True)
    .by_duration('==', (1, 16), preprolated=True)
    .by_contiguity()
    .by_leaves()
    [0]
    .flatten()"
),

staccati=consort.AttachmentExpression(
    attachments=indicatortools.Articulation('staccato'),
    selector=selectortools.Selector()
    .by_logical_tie(pitched=True)
    .by_duration('==', (1, 16), preprolated=True)
    [0],
    .flatten()
),

tremoli=consort.AttachmentExpression(
    attachments=spannertools.StemTremoloSpanner(),
    selector=selectortools.Selector()
    .by_logical_tie(pitched=True)
    .by_duration('>=' , (1, 8), preprolated=True)
    .by_contiguity()
    .by_leaves()
    [0],
    .flatten()
),

color='magenta',
labels=[],
pitch_handler=consort.PitchClassPitchHandler(
    forbid_repetitions=True,
    leap_constraint=12,
    logical_tie_expressions=[
        None,
        consort.ChordExpression([0, 3]),
        None,
        None,
        consort.ChordExpression([0, 5]),
    ],
pitch_specifier=abbreviations.agitato_pitch_specifier,
register_specifier=consort.RegisterSpecifier(  
    base_pitch='A2',
    phrase_inflections=consort.RegisterInflection
    .zigzag(6)
    .reverse()
    .align(),
    segment_inflections=consort.RegisterInflection
    .descending(width=12)
    .align()
    ),
)

rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
burnish_specifier = rhythm makertools.BurnishSpecifier(
    left_classes=[scoretools.Rest],
    left_counts=[1, 0],
    right_classes=[scoretools.Rest],
    right_counts=[1],
),
extra_counts_per_division=[0, 0, 1],
output_masks=[
    rhythm makertools.SustainMask(
        indices=[0],
        period=3,
    ),
],
talea = rhythm makertools.Talea(
    counts=[1, 1, 1, 2],
    denominator=16,
),
)

D.2.4  ersilia.materials.guitar_continuo_musicSpecifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythm makertools
from abjad.tools import scoretools
from abjad.tools import selectortools
from abjad.tools import spannertools

guitar_continuo_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector()
            .by_leaves()
            .by_run(scoretools.Note)
            .by_counts(
                [3, 3, 4],
                cyclic=True,
                overhang=True,
                fuse_overhang=True,
            )
            [0],
        ),
    ),
    dynamic_expressions=consort.DynamicExpression(
        start_dynamic_tokens='p mp',
        only_first=True,
    ),
    slur=consort.AttachmentExpression(
        attachments=spannertools.Slur(),
        selector=selectortools.Selector()
        .by_leaves()
D.2.5 ersilia.materials.guitar_pointillist_harmonics_music_specifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import rhythmmakertools
from abjad.tools import scoretools
from ersilia.materials import abbreviations

guitar_pointillist_harmonics_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expressions=consort.DynamicExpression(
            dynamic_tokens='p mp',
        ),
    ),
    color=None,
    labels=[],
    pitch_handler=consort.PitchClassPitchHandler(
        forbid_repetitions=True,
        leap_constraint=6,
        logical_tie_expressions=[
            consort.HarmonicExpression('P4'),
            consort.HarmonicExpression('P5'),
            consort.HarmonicExpression('P8'),
            consort.HarmonicExpression('P5'),
        ],
        pitchSpecifier=abbreviations.agitato_pitch specifier,
    )
)
```
```
registerSpecifier = consort.RegisterSpecifier(
    base_pitch='E2',
    phrase_inflections=consort.RegisterInflection
        .zigzag(6)
        .reverse()
        .align(),
    segment_inflections=consort.RegisterInflection
        .descending(width=12)
        .align()
),
    rhythm_maker=consort.CompositeRhythmMaker(
        default=rhythmmakertools.EvenDivisionRhythmMaker(
            burnish_specifier=rhythmmakertools.BurnishSpecifier(
                left_classes=[scoretools.Rest],
                left_counts=[1, 0],
                right_classes=[scoretools.Rest],
                right_counts=[1],
            ),
            denominators=[8, 8, 16],
            extra_counts_per_division=[0, 0, 1, 0, 1, 2],
        ),
        last=rhythmmakertools.IncisedRhythmMaker(
            incise_specifier=rhythmmakertools.InciseSpecifier(
                fill_with_notes=False,
                prefix_counts=[1],
                prefix_talea=[1],
                talea_denominator=8,
            ),
        ),
    )
)
```
is_once=True,  
    property_path='transparent',  
    value=True,  
),  
markuptools.Markup.musicglyph('scripts.coda'),  
indicatortools.Articulation('accent'),  
indicatortools.Dynamic('sfz'),  
],  
),  
is_destructive=True,  
selector=selectortools.Selector()  
    .by_logical_tie(pitched=True)  
    .by_contiguity()  
    .by_length('>', 1)  
    .by_leaves()  
    [-1]  
),  
dynamic_expressions=consort.DynamicExpression(  
    dynamic_tokens='p ppp p ppp mf p',  
    only_first=True,  
),  
laissez_vibrer=abbreviations.laissez_vibrer,  
),  
pitch_handler=consort.AbsolutePitchHandler(  
    logical_tie_expressions=[  
        consort.ChordExpression(  
            chord_expr=_,  
            arpeggio_direction=Center,  
        ) for _ in abbreviations.guitar_chords  
    ],  
),  
rhythm_maker=rhythmmakertools.IncisedRhythmMaker(  
    inciseSpecifier=rhythmmakertools.InciseSpecifier(  
        fill_with_notes=False,  
        prefix_counts=[1, 1, 1, 2, 1, 2, 3],  
        prefix_talea=[1],  
        talea_denominator=16,  
    ),  
),  
)

D.2.7 ersilia.materials.guitar_tremolo_musicSpecifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

guitar_tremolo_musicSpecifier = Consort.MusicSpecifier(  
    attachment_handler=consort.AttachmentHandler(  
    )  
)
```python
accents = consort.AttachmentExpression(
    attachments=indicatortools.Articulation('accent'),
    selector=selectortools.select_pitched_runs()
    .by_counts([3], cyclic=True)

    ),

dynamic_expressions = consort.DynamicExpression(
    division_period=2,
    dynamic_tokens='pp mf p p mf pp',
    start_dynamic_tokens='fp o',
    stop_dynamic_tokens='o f',
)

stem_tremolo_spanner = consort.AttachmentExpression(
    attachments=spannertools.StemTremoloSpanner(),
    selector=selectortools.select_pitched_runs(),
)

color = 'red',
pitch_handler = consort.AbsolutePitchHandler(
    logical_tie_expressions=[
        consort.ChordExpression(
            chord_expr=_,
            ) for _ in abbreviations.guitar_chords
    ],

    pitch_specifier = consort.PitchSpecifier(
        pitch_segments=(
            'D3',
            'F3',
            'G2',
            ),
        ratio=(1, 1, 1),
    ),

    rhythm_maker = rhythmmakertools.NoteRhythmMaker(
        tie_specifier=rhythmmakertools.TieSpecifier(
            tie_across_divisions=True,
        ),
    ),
)
```

D.2.8 ersilia.materials.guitar_undulation_tremolo_musicSpecifier

```python
# -*- encoding: utf-8 -*-

import consort

from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools

from ersilia.materials import abbreviations


guitar_undulation_tremolo_musicSpecifier = consort.MusicSpecifier(

    attachment_handler = consort.AttachmentHandler(
        accents=consort.AttachmentExpression(

```

909
attachments=indicatortools.Articulation('accent'),
selector=selectortools.Selector()
  .by_logical_tie(pitched=True)
  .by_duration('==', (1, 8), preprolated=True)
[0]
  .with_next_leaf()
),
dynamic_expressions=consort.DynamicExpression(
  dynamic_tokens='p mp pp',
  start_dynamic_tokens='o',
  stop_dynamic_tokens='o',
  division_period=2,
),
stem_tremolo_spanner=consort.AttachmentExpression(
  attachments=spanntools.StemTremoloSpanner(),
  selector=selectortools.select_pitched_runs(),
),
color='red',
lables=[],
pitch_handler=consort.AbsolutePitchHandler(
  forbid_repetitions=True,
  logical_tie_expressions=(
    consort.ChordExpression(chord_expr=[0, 7, 14, 15]),
  ),
  pitch_specifier=abbreviations.agitato_pitch_specifier,
  pitch_operation_specifier=abbreviations.pitch_operation_specifier,
),
rhythm_maker=rhythmmakertools.EvenDivisionRhythmMaker(
  denominators=[8],
  extra_counts_per_division=[0, 1],
  output_masks=[
    rhythmmakertools.SustainMask(
      indices=[2],
      period=3,
    ),
    rhythmmakertools.SustainMask(
      indices=[0, -1],
    ),
  ],
  tie_specifier=rhythmmakertools.TieSpecifier(
    tie_across_divisions=True,
  ),
)

D.2.9 ersilia.materials.percussion_bamboo_windchimes_musicSpecifier

# -*- encoding: utf-8 -*-
import consort
import ersilia
from abjad.tools import indicatortools
from abjad.tools import selectortools
from abjad.tools import rhythmmakertools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

percussion_bamboo_windchimes_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=[
                indicatortools.Articulation('accent'),
                indicatortools.Dynamic('f'),
            ],
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('==', (1, 16), preprolated=True)
            [0],
        ),
        piano=consort.AttachmentExpression(
            attachments=indicatortools.Dynamic('pp'),
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('>', (1, 16), preprolated=True)
            [0]
        ),
        text_spanner=consort.AttachmentExpression(
            attachments=abbreviations.make_text_spanner('windchimes'),
            selector=selectortools.select_pitched_runs(),
        ),
        tremolo=consort.AttachmentExpression(
            attachments=spannertools.StemTremoloSpanner(),
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('>', (1, 16), preprolated=True)
            [0],
        ),
        color='yellow',
        labels=['bamboo windchimes'],
    pitch_handler=consort.AbsolutePitchHandler(
        pitchSpecifier=ersilia.Percussion.BAMBOO_WIND_CHIMES,
        pitchesAreNonsemantic=True,
    ),
    rhythm_maker=consort.CompositeRhythmMaker(
        default=rhythmmakertools.NoteRhythmMaker(
            tieSpecifier=rhythmmakertools.TieSpecifier(
                tieAcrossDivisions=True,
            ),
        ),
        first=rhythmmakertools.IncisedRhythmMaker(
            inciseSpecifier=rhythmmakertools.InciseSpecifier(
                fillWithNotes=False,
                prefixCounts=[1],
                prefixTalea=[1],
                taleaDenominator=16,
            ),
        ),
    )}
D.2.10 ersilia.materials.percussion_crotales_flash_musicSpecifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from ersilia.materials import abbreviations

percussion_crotales_flash_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        clef_spanner=consort.ClefSpanner('treble^15'),
        dynamic_expressions=consort.DynamicExpression(
            only_first=True,
            start_dynamic_tokens='f p mp',
        ),
        text_spanner=consort.AttachmentExpression(
            attachments=abbreviations.make_text_spanner('crotales'),
            selector=selectortools.select_pitched_runs(),
        ),
    ),
    color=None,
    labels=[],
    pitch_handler=consort.PitchClassPitchHandler(
        forbid_repetitions=True,
        pitch_specifier=abbreviations.agitato_pitch_specifier,
        register_specifier=consort.RegisterSpecifier(
            base_pitch="c'''",
            segment_inflections=consort.RegisterInflection.zigzag(6).
        ),
    ),
    rhythm_maker=rhythmmakertools.IncisedRhythmMaker(
        incise_specifier=rhythmmakertools.InciseSpecifier(
            fill_with_notes=False,
            prefix_counts=[4, 3],
            prefix_talea=[1],
            talea_denominator=16,
        ),
    )
)
```

D.2.11 ersilia.materials.percussion_crotales_interruption_musicSpecifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools

```

912
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

percussion_crotales_interruption_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=[
                indicatortools.Articulation('accent'),
                indicatortools.Dynamic('fff'),
            ],
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('==', (1, 16), preprolated=True)
                [0],
        ),
        clef_spanner=consort.ClefSpanner('treble^15'),
        shimmer=consort.AttachmentExpression(
            attachments=[
                indicatortools.Articulation('accent'),
                indicatortools.Dynamic('fp'),
            ],
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('>', (1, 16), preprolated=True)
                .by_leaves()
                .by_length('==', 1)
                [0]
        ),
        swell=consort.AttachmentExpression(
            attachments=spannertools.Hairpin('niente < f'),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('>', (1, 16), preprolated=True)
                .by_leaves()
                .by_length('>', 1)
        ),
        text_spanner=consort.AttachmentExpression(
            attachments=abbreviations.make_text_spanner('crotales'),
            selector=selectortools.select_pitched_runs(),
        ),
        tremolo=consort.AttachmentExpression(
            attachments=spannertools.StemTremoloSpanner(),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('>', (1, 16), preprolated=True)
        ),
    ),
    color='yellow',
)
labels=[],
pitch_handler=consort.PitchClassPitchHandler(
    forbid_repetitions=True,
    pitch_specifier=abbreviations.agitato_pitch_specifier,
    register_specifier=consort.RegisterSpecifier(
        base_pitch="c'\"",
        segment_inflections=consort.RegisterInflection
        .zigzag(6)
        .align()
    ),
),
rhythm_maker=consort.CompositeRhythmMaker(
    default=rhythmmakertools.NoteRhythmMaker(
        tie_specifier=rhythmmakertools.TieSpecifier(
            tie_across_divisions=True,
        ),
    ),
    first=rhythmmakertools.IncisedRhythmMaker(
        incise_specifier=rhythmmakertools.InciseSpecifier(
            fill_with_notes=False,
            prefix_counts=[1],
            prefix_talea=[1],
            talea_denominator=16,
        ),
    ),
),

D.2.12 ERSILIA.MATERIALS.PERCUSION_LOW_PEDAL_MUSICSpecifier

# -*- encoding: utf-8 -*-
import consort
import ersilia
from abjad.tools import indicatortools
from abjad.tools import pitchtools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

percussion_low_pedal_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector()
            .by_logical_tie()
            .get_slice(start=1, apply_to_each=False)
            [0]
        ),
    ),
bass_drum_indication=consort.AttachmentExpression(
        attachments=abbreviations.make_text_spanner('bass drum'),
        selector=selectortools.select_pitched_runs(),
    )
selector = selectortools.Selector()
    .by_logical_tie()
    .by_pitch(pitches=ersilia.Percussion.BASS_DRUM)
    .by_contiguity()
    .by_leaves()

tam_tam_indication = consort.AttachmentExpression(
    attachments=consort.AttachmentExpression(
        attachments=abbreviations.make_text_spanner('tam'),
        selector=selectortools.select_pitched_runs(),
    ),
    selector=selectortools.Selector()
    .by_logical_tie()
    .by_pitch(pitches=ersilia.Percussion.TAM_TAM)
    .by_contiguity()
    .by_leaves()
)

dynamic_expressions = consort.DynamicExpression(
    division_period=2,
    dynamic_tokens='p ppp p ppp mf',
    start_dynamic_tokens='o',
    stop_dynamic_tokens='o',
)

stem_tremolo_spanner = consort.AttachmentExpression(
    attachments=spannertools.StemTremoloSpanner(),
    selector=selectortools.select_pitched_runs(),
)

color = 'red',
labels=[]
pitch_handler = consort.AbsolutePitchHandler(
    pitch_application_rate='phrase',
    pitchSpecifier=pitchtools.PitchSegment([
        ersilia.Percussion.BASS_DRUM,
        ersilia.Percussion.TAM_TAM,
    ]),
    pitches_are_nonsemantic=True,
)

rhythm_maker = rhythmmakertools.EvenDivisionRhythmMaker(
    denominators=[8],
    output_masks=[
        rhythmmakertools.SustainMask(
            indices=[0, 1],
            period=3,
        ),
        rhythmmakertools.SustainMask(
            indices=[0, -1],
        ),
    ],
    tieSpecifier=rhythmmakertools.TieSpecifier(
        tie_across_divisions=True,
    ),
)
D.2.13 ersilia.materials.percussion_marimba_agitato_musicSpecifier

# -*- encoding: utf-8 -*-
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations
import consort

percussion_marimba_agitato_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.select_pitched_runs()[0],
        ),
        chords=consort.AttachmentExpression(
            attachments=[
                consort.ChordExpression(chord_expr=[0, 3]),
                consort.ChordExpression(chord_expr=[0, 5]),
            ],
            is_destructive=True,
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('==', (1, 16), preprolated=True)
                .by_pattern(
                    rhythmmakertools.BooleanPattern(
                        indices=[0, 3],
                        period=7,
                        ),
                ),
            ),
        clef_spanner=consort.ClefSpannerExpression(),
        dynamic_expressions=consort.DynamicExpression(
            division_period=2,
            dynamic_tokens='mf mp fff',
            start_dynamic_tokens='f',
            stop_dynamic_tokens='mf',
        ),
        staccati=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('staccato'),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('==', (1, 16), preprolated=True)
                [0]
        ),
        staff_lines_spanner=spannertools.StaffLinesSpanner([-4, -2, 0, 2, 4]),
        text_spanner=consort.AttachmentExpression(
            attachments=abbreviations.make_text_spanner('marimba'),
            selector=selectortools.select_pitched_runs(),
        ),
    ),
)
tremolo_chords=consort.AttachmentExpression(
    attachments=[
        spannertools.StemTremoloSpanner(),
        consort.ChordExpression(chord_expr=[0, 3]),
    ],
    is_destructive=True,
    selector=selectortools.Selector()
        .by_logical_tie(pitched=True)
        .by_duration('>', (1, 16), preprolated=True)
    ),
    color='magenta',
    labels=[],
    pitch_handler=consort.PitchClassPitchHandler(
        forbid_repetitions=True,
        leap_constraint=9,
        pitch_specifier=abbreviations.agitato_pitch_specifier,
        register_specifier=consort.RegisterSpecifier(
            base_pitch='F2',
            phrase_inflections=consort.RegisterInflection.zigzag(6)
                .reverse()
                .align(),
            segment_inflections=consort.RegisterInflection
                .descending(width=12)
                .align()
        ),
        register_spread=6,
    ),
    rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
        extra_counts_per_division=[0, 0, 1, 2, 0, 1],
        output_masks=[
            rhythmmakertools.SustainMask(
                indices=[1],
                period=3,
            ),
        ],
        talea=rhythmmakertools.Talea(
            counts=[
                1, -1,
                1, 1, -1,
                1, 1, 1, -1,
                1, 1, -1,
                1, 1, 1, -2,
                1,
            ],
            denominator=16,
        ),
    )
)
D.2.14 ersilia.materials.percussion_marimba_ostinato_musicSpecifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

percussion_marimba_ostinato_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        clef_spanner=consort.ClefSpannerExpression(),
        deadstroke=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('stopped'),
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('==', (1, 16), preprolated=True)
            .by_contiguity()
            .by_length('==', 1)
            .by_leaves()
        ),
        dynamic_expressions=consort.DynamicExpression(
            dynamic_tokens='p',
        ),
        slur=consort.AttachmentExpression(
            attachments=spannertools.Slur(),
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('==', (1, 16), preprolated=True)
            .by_contiguity()
            .by_length('>', 1)
            .by_leaves()
        ),
        staccati=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('staccato'),
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('==', (1, 16), preprolated=True)
            .by_contiguity()
            .by_length('>', 1)
            .by_leaves()
        ),
        staff_lines_spanner=spannertools.StaffLinesSpanner([-4, -2, 0, 2, 4]),
        text_spanner=consort.AttachmentExpression(
            attachments=abbreviations.make_text_spanner('marimba'),
            selector=selectortools.select_pitched_runs(),
        ),
        color='darkyellow',
    pitch_handler=consort.AbsolutePitchHandler(
```

918
forbid_repetitions=True,
logical_tie_expressions=[
    consort.ChordExpression(chord_expr=[0, 5]),
],
pitch_specifier="d' f'",
),
rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
    extra_counts_per_division=[0, 0, 1, 2, 0, 1],
talea=rhythmmakertools.Talea(
    counts=[1, 1, -3],
    denominator=16,
),
),
D.2.15 ersilia.materials.percussion_marimba_tremolo_musicSpecifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import pitchtools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

percussion_marimba_tremolo_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        clef_spanner=consort.ClefSpannerExpression(),
        dynamic_expressions=consort.DynamicExpression(
            division_period=2,
            dynamic_tokens='p ppp',
            start_dynamic_tokens='o',
            stop_dynamic_tokens='o',
        ),
        staff_lines_spanner=spannertools.StaffLinesSpanner([-4, -2, 0, 2, 4]),
        stem_tremolo_spanner=consort.AttachmentExpression(
            attachments=spannertools.StemTremoloSpanner(),
            selector=selectortools.select_pitched_runs(),
        ),
        text_spanner=consort.AttachmentExpression(
            attachments=abbreviations.make_text_spanner('marimba'),
            selector=selectortools.select_pitched_runs(),
        ),
        color='red',
        labels=[],
    ),
pitch_handler=consort.PitchClassPitchHandler(
    leap_constraint=9,
    logical_tie_expressions=(
        consort.ChordExpression(
            chord_expr=pitchtools.IntervalSegment([0, 3]),
        ),
    ),
)
chord_expr=pitchtools.IntervalSegment([0, 5]),
    ),
    consort.ChordExpression(
        chord_expr=pitchtools.IntervalSegment([0, 3]),
    ),
    consort.ChordExpression(
        chord_expr=pitchtools.IntervalSegment([0, 2]),
    ),
    pitch_specifier=consort.PitchSpecifier(
        pitch_segments=[
            "d d f d f g",
            "ef gf df b df",
        ],
    ),
    register_specifier=consort_RegisterSpecifier(
        base_pitch='F2',
        segment_inflections=consort.RegisterInflection
            .zigzag(12)
            .reverse()
            .align(),
    register_spread=3,
    ),
    rhythm_maker=rhythmmakertools.EvenDivisionRhythmMaker(
        denominators=[16],
        output_masks=[
            rhythmmakertools.SustainMask(
                indices=[2],
                invert=True,
                period=3,
            ),
        ],
    tieSpecifier=rhythmmakertools.TieSpecifier(
        tie_across_divisions=True,
    ),
)
attachments=[
    [
        indicatorTools.Articulation('accent'),
        indicatorTools.Dynamic('fff'),
    ],
],
selector=selectortools.Selector()
    .by_logical_tie(pitched=True)
    .by_duration('==', (1, 16), preprolated=True)
[0],
),
shimmer=consort.AttachmentExpression(
    attachments=[
        [
            indicatorTools.Articulation('accent'),
            indicatorTools.Dynamic('fff'),
        ],
    ],
selector=selectortools.Selector()
    .by_logical_tie(pitched=True)
    .by_duration('==', (1, 16), preprolated=True)
[0],
),
swell=consort.AttachmentExpression(
    attachments=spannertools.Hairpin('niente < f'),
selector=selectortools.Selector()
    .by_logical_tie(pitched=True)
    .by_duration('>', (1, 16), preprolated=True)
    .by_leaves()
    .by_length('==', 1)
[0]
),
text_spanner=consort.AttachmentExpression(
    attachments=abbreviations.make_text_spanner('snare'),
selector=selectortools.select_pitched_runs(),
),
tremolo=consort.AttachmentExpression(
    attachments=spannertools.StemTremoloSpanner(),
selector=selectortools.Selector()
    .by_logical_tie(pitched=True)
    .by_duration('>', (1, 16), preprolated=True)
),
),
color='yellow',
labels=[],
pitch_handler=consort.AbsolutePitchHandler(  
pitch_specifier=ersilia.Percussion.SNARE_DRUM,  
pitches_are_nonsemantic=True,  
),
rhythm_maker=consort.CompositeRhythmMaker(  
default=rhythmmakertools.NoteRhythmMaker(  
tie_specifier=rhythmmakertools.TieSpecifier(  
tie_across_divisions=True,
IncisedRhythmMaker(
    incise_specifier=rhythmmakertools.InciseSpecifier(
        fill_with_notes=False,
        prefix_counts=[1, 2],
        prefix_talea=[1, -1],
        talea_denominator=16,
    ),
),
),

D.2.17 ersilia.materials.percussion_temple_block_fanfare_music_specifier

# -*- encoding: utf-8 -*-
import consort
import ersilia
from abjad.tools import pitchtools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import indicatortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

percussion_temple_block_fanfare_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accent=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('>', (1, 16), preprolated=True)
            [0]
        ),
        chords=consort.AttachmentExpression(
            attachments=(
                consort.ChordExpression(
                    chord_expr=pitchtools.PitchSegment([
                        ersilia.Percussion.WOOD_BLOCK_5,
                        ersilia.Percussion.WOOD_BLOCK_4,
                    ]),
                ),
                None,
                consort.ChordExpression(
                    chord_expr=pitchtools.PitchSegment([
                        ersilia.Percussion.WOOD_BLOCK_4,
                        ersilia.Percussion.WOOD_BLOCK_3,
                    ]),
                ),
                consort.ChordExpression(
                    chord_expr=pitchtools.PitchSegment([
                        ersilia.Percussion.WOOD_BLOCK_3,
                        ersilia.Percussion.WOOD_BLOCK_2,
                    ]),
                ),
            ),
    ),
)
None,
    consort.ChordExpression(
        chord_expr=pitchtools.PitchSegment([
            ersilia.Percussion.WOOD_BLOCK_2,
            ersilia.Percussion.WOOD_BLOCK_1,
        ]),
        is_destructive=True,
        selector=selectortools.Selector()
        .by_logical_tie(pitched=True)
        .by_duration('>', (1, 16), preprolated=True)
    ),
    dynamic_expression=consort.DynamicExpression(
        division_period=2,
        start_dynamic_tokens='p fp',
        stop_dynamic_tokens='f',
        unsustained=True,
    ),
    staccato=consort.AttachmentExpression(
        attachments=indicatortools.Articulation('staccato'),
        selector=selectortools.Selector()
        .by_logical_tie(pitched=True)
        .by_duration('<=', (1, 16), preprolated=True)
        [0],
    ),
    staff_lines_spanner=spannertools.StaffLinesSpanner([-4, -2, 0, 2, 4]),
    text_spanner=consort.AttachmentExpression(
        attachments=abbreviations.make_text_spanner('blocks'),
        selector=selectortools.select_pitched_runs(),
    ),
    tremolo=consort.AttachmentExpression(
        attachments=spannertools.StemTremoloSpanner(),
        selector=selectortools.Selector()
        .by_logical_tie(pitched=True)
        .by_duration('>', (1, 16), preprolated=True)
    ),
    color='magenta',
    pitch_handler=consort.AbsolutePitchHandler(
        #logical_tie_expressions=[
        # ],
    pitchSpecifier=pitchtools.PitchSegment([
        ersilia.Percussion.WOOD_BLOCK_5,
        ersilia.Percussion.WOOD_BLOCK_4,
        ersilia.Percussion.WOOD_BLOCK_3,
        ersilia.Percussion.WOOD_BLOCK_2,
        ersilia.Percussion.WOOD_BLOCK_1,
        ersilia.Percussion.WOOD_BLOCK_4,
        ersilia.Percussion.WOOD_BLOCK_3,
        ersilia.Percussion.WOOD_BLOCK_2,
        ersilia.Percussion.WOOD_BLOCK_3,
D.2.18 ersilia.materials.percussion_tom_fanfare_musicSpecifier

    ersilia.Percussion.WOOD_BLOCK_2,
    ersilia.Percussion.WOOD_BLOCK_2,
    ),
pitch_operationSpecifier=consort.PitchOperationSpecifier(
    pitch_operations=[
        None,
        pitchtools.Retrogression(),
        None,
    ],
    pitches_are_nonsemantic=True,
),
rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
    extra_counts_per_division=[0, 1, 2],
talea=rhythmmakertools.Talea(
        counts=[
            1, 1, -1,
            1, 1, -1,
            1, 1, -2,
            1, 1, 1, 1, 1, 1, -1,
            1, 1, -2,
            1, 1, -1,
            1, 1, 1, 1, 1, 1, -1,
            1, 1, 1, -1,
            1, 1, 1, 1, -2,
        ],
        denominator=16,
    ),
    output_masks=[
        rhythmmakertools.SustainMask(
            indices=[1],
            period=2,
        ),
    ],
)
attachments=indicatortools.Articulation('accent'),
selector=selectortools.Selector()
    .by_logical_tie(pitched=True)
    .by_duration('>', (1, 16), preprolated=True)
[0],

dynamic_expression=consort.DynamicExpression(
    start_dynamic_tokens='p fp',
    stop_dynamic_tokens='f',
    unsustained=True,
),
staccato=consort.AttachmentExpression(
    attachments=indicatortools.Articulation('staccato'),
    selector=selectortools.Selector()
    .by_logical_tie(pitched=True)
    .by_duration('<=', (1, 16), preprolated=True)
[0],
)

staff_lines_spanner=spannertools.StaffLinesSpanner([-4, -2, 0, 2, 4]),
text_spanner=consort.AttachmentExpression(
    attachments=abbreviations.make_text_spanner('toms'),
    selector=selectortools.select_pitched_runs(),
),
tremolo=consort.AttachmentExpression(
    attachments=spannertools.StemTremoloSpanner(),
    selector=selectortools.Selector()
    .by_logical_tie(pitched=True)
    .by_duration('>', (1, 16), preprolated=True)
),

color='magenta',
pitch_handler=consort.AbsolutePitchHandler(
    logical_tie_expressions=[
        #
    ],
pitch_specifier=pitchtools.PitchSegment([
ersilia.Percussion.TOM_4,
ersilia.Percussion.TOM_3,
ersilia.Percussion.TOM_2,
ersilia.Percussion.TOM_1,
ersilia.Percussion.TOM_4,
ersilia.Percussion.TOM_3,
ersilia.Percussion.TOM_2,
ersilia.Percussion.TOM_3,
ersilia.Percussion.TOM_2,
ersilia.Percussion.TOM_2,
]),
pitch_operation_specifier=consort.PitchOperationSpecifier(
    pitch_operations=[
        None,
pitchtools.Retrogression(),
        None,
    ],
pitches_are_nonsemantic=True,
D.2.19 ersilia.materials.permitted_time_signatures

# -*- encoding: utf-8 -*-
from abjad import *

permitted_time_signatures = indicatortools.TimeSignatureInventory([
    #(2, 4),
    (4, 8),
    (5, 8),
    (6, 8),
    #(3, 4),
    (7, 8),
    (4, 4),
    #(9, 8),
    #(5, 4),
    #(6, 4),
])

D.2.20 ersilia.materials.piano_agitato_music_specifier

# -*- encoding: utf-8 -*-
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import scoretools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations
import consort

piano_agitato_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('==', (1, 8), preprolated=True)
                [0],
        ),
        dynamic_expressions=consort.DynamicExpression(
            division_period=2,
            dynamic_tokens='mf mp fff',
            start_dynamic_tokens='f',
            stop_dynamic_tokens='mf',
        ),
        mordent=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('mordent'),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('>=', (1, 8), preprolated=True)
                .by_class(scoretools.Note)
                [0],
        ),
        staccati=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('staccato'),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('==', (1, 16), preprolated=True)
                [0],
        ),
        tremoli=consort.AttachmentExpression(
            attachments=spannertools.StemTremoloSpanner(),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('>=', (1, 8), preprolated=True)
                .by_class(scoretools.Chord)
                [0],
        ),
        color='magenta',
        labels=[],
    pitch_handler=consort.PitchClassPitchHandler(
        forbid_repetitions=True,
        leap_constraint=12,
        logical_tie_expressions=[
            consort.ChordExpression([-6, -3, 3, 8]),
            None,
            consort.ChordExpression([0, 3]),
            None,
            None,
            consort.ChordExpression([0, 3]),]}

927
consort.ChordExpression([-1, 2]),
None,
consort.KeyClusterExpression(
    include_black_keys=False,
),
consort.ChordExpression([0, 3]),
None,
None,
],
pitch_specifier=abbreviations.agitato_pitch_specifier,
register_specifier=consort.RegisterSpecifier(
    base_pitch='G3',
    phrase_inflections=consort.RegisterInflection.zigzag(6)
        .reverse().align(),
    segment_inflections=consort.RegisterInflection.descending(
        width=12).align()
),
register_spread=6,
),

rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
    burnish_specifier=rhythmmakertools.BurnishSpecifier(
        left_classes=[scoretools.Rest],
        left_counts=[1, 1, 0],
        right_classes=[scoretools.Rest],
        right_counts=[1],
    ),
    extra_counts_per_division=[0, 0, 1],
    output_masks=[
        rhythmmakertools.SustainMask(
            indices=[0],
            period=3,
        ),
    ],
    talea=rhythmmakertools.Talea(
        counts=[1, 1, 1, 1, 2],
        denominator=16,
    ),
),
)

D.2.21  ersilia.materials.piano_arm_cluster_music_specifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools

piano_arm_cluster_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
        ),
        selector=selectortools.Selector()
)
.by_logical_tie(pitched=True)[0],
},
dynamic_expressions=consort.AttachmentExpression(
    attachments=indicatorools.Dynamic('fff'),
    selector=selectorools.Selector()
    .by_logical_tie(pitched=True)
    [0]
},
color='yellow',
labels=['piano arm cluster'],
pitch_handler=consort.AbsolutePitchHandler(
    logical_tie_expressions=[
        consort.KeyClusterExpression(
            staff_space_width=19,
        ),
    ],
),
rhythm_maker=rhythmmakertools.IncisedRhythmMaker(
    inciseSpecifier=rhythmmakertools.InciseSpecifier(
        fill_with_notes=False,
        prefix_counts=[1],
        prefix_talea=[1],
        talea_denominator=8,
    ),
),
)

D.2.22  ersilia.materials.piano_glissando_musicSpecifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import rhythmmakertools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

piano_glissando_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expressions=consort.DynamicExpression(
            dynamic_tokens='p',
            only_first=True,
        ),
        glissando=spannertools.Glissando(),
        keys_spanner=(
            abbreviations.chromatic_keys_spanner,
            abbreviations.white_keys_spanner,
            abbreviations.white_keys_spanner,
        ),
    ),
    color=None,
    labels=[],
pitch_handler=consort.AbsolutePitchHandler(
    forbid_repetitions=True,
D.2.23   ersilia.materials.piano_palm_cluster_musicSpecifier

    pitch_specifier="c' c'' f' f'' c' f' c'' c'' f'' c'' f''", 
    rhythm_maker=consort.CompositeRhythmMaker( 
        last=abjad.rhythmmakertools.IncisedRhythmMaker( 
            incise_specifier=abjad.rhythmmakertools.InciseSpecifier( 
                prefix_counts=[0], 
                suffix_talea=[1], 
                suffix_counts=[1], 
                talea_denominator=16, 
            ), 
            duration_spelling_specifier=abjad.rhythmmakertools.DurationSpellingSpecifier( 
                forbidden_written_duration=(1, 4), 
                forbid_meter_rewriting=True, 
                spell_metrically='unassignable', 
            ), 
            tie_specifier=abjad.rhythmmakertools.TieSpecifier( 
                strip_ties=True, 
            ), 
        ), 
        default=abjad.rhythmmakertools.EvenDivisionRhythmMaker( 
            denominators=(4,), 
            duration_spelling_specifier=abjad.rhythmmakertools.DurationSpellingSpecifier( 
                forbidden_written_duration=(1, 4), 
                forbid_meter_rewriting=True, 
                spell_metrically='unassignable', 
            ), 
        ), 
    ), 
    piano_palm_cluster_musicSpecifier = consort.MusicSpecifier( 
        attachment_handler=consort.AttachmentHandler( 
            dynamic_expressions=consort.DynamicExpression( 
                dynamic_tokens='p mf mp pp f', 
                division_period=2, 
            ), 
            octavation=consort.OctavationExpression(), 
        ), 
        labels='pedaled', 
        pitch_handler=consort.PitchClassPitchHandler( 
            forbid_repetitions=True, 
            logical_tie_expressions={ 
                consort.KeyClusterExpression( 
                    include_black_keys=False, 
                ), 
            }, 
        ), 
    )
include_black_keys=False,
  ),
  consort.KeyClusterExpression(
    staff_space_width=9,
  ),
  pitch_specifier="c e g",
  registerSpecifier=consort/RegisterSpecifier(
    division_inflections=consort/RegisterInflection.descending(),
    phrase_inflections=consort/RegisterInflection.zigzag(),
    segment_inflections=consort/RegisterInflection.descending(
      width=24).align()
  ),
  rhythm_maker=rhythmmakertools/EvenDivisionRhythmMaker(
    burnish_specifier=rhythmmakertools/BurnishSpecifier(
      left_classes=[scoretools/Rest],
      left_counts=[1],
      right_classes=[scoretools/Rest],
      right_counts=[1, 0, 0],
    ),
    denominators=[16, 16, 8, 16, 16, 8],
    extra_counts_per_division=(0, 0, 1),
  ),
)

D.2.24 ersilia.materials.piano_pedals_music_setting

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import rhythmmakertools

piano_pedals_music_setting = consort/MusicSetting(
  timespan_maker=consort/DependentTimespanMaker(
    hysteresis=(1, 4),
    include_inner_starts=True,
    include_inner_stops=False,
    inspect_music=True,
    labels=(
      'pedaled',
    ),
    voice_names=(
      'Piano Upper Voice',
      'Piano Lower Voice',
    ),
  ),
  piano_pedals=consort/MusicSpecifier(
    attachment_handler=consort/AttachmentHandler(
      piano_pedal_spanner=consort/ComplexPianoPedalSpanner(
        include_inner_leaves=True,
      ),
    ),
  rhythm_maker=rhythmmakertools/SkipRhythmMaker
duration_spelling_specifier=rhythmmakertools.DurationSpellingSpecifier(
    forbid_meter_rewriting=True,
),
)

D.2.25  ersilia.materials.piano_pointillist_musicSpecifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import scoretools
from abjad.tools import selectortools
from ersilia.materials import abbreviations

piano_pointillist_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expressions=consort.DynamicExpression(
            start_dynamic_tokens='ppp',
            only_first=True,
        ),
        mordent=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('mordent'),
            selector=selectortools.Selector() .by_class(scoretools.Note) .by_logical_tie()
        ),
        tenuti=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('tenuto'),
            selector=selectortools.Selector() .by_logical_tie(pitched=True)
        ),
    ),
    color='darkyellow',
    labels=['pedaled'],
    pitch_handler=consort.PitchClassPitchHandler(
        logical_tie_expressions=(
            None,
            consort.ChordExpression([-2, 3]),
            consort.ChordExpression([0, 3]),
            None,
            consort.ChordExpression([-4, 5]),
        ),
        pitchSpecifier=abbreviations.agitato_pitchSpecifier,
        registerSpecifier=consort.RegisterSpecifier(
            basePitch='G3',
            phraseInflections=consort.RegisterInflection(zigzag(12))
        )
    )
)

932
.reverse()
    .align(),
    segment_inflctions=consort.RegisterInflection
        .descending(width=12)
        .align()
    ),
    register_spread=6,
)
	rhythm_maker=consort.CompositeRhythmMaker(
    default=rhythmmakertools.TaleaRhythmMaker(
        extra_counts_per_division=[0, 0, 1],
        talea=rhythmmakertools.Talea(
            counts=[1, -2, 1, -3, 1, -4],
            denominator=16,
        ),
    ),
    last=rhythmmakertools.IncisedRhythmMaker(
        incise_specifier=rhythmmakertools.InciseSpecifier(
            fill_with_notes=False,
            prefix_counts=[2],
            prefix_talea=[-1, 1, 1],
            talea_denominator=16,
        ),
    ),
)

D.2.26 ersilia.materials.piano_string_glissando_musicSpecifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import rhythmmakertools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

piano_string_glissando_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        clef_spanner=consort.ClefSpanner('percussion'),
        dynamic_expressions=consort.DynamicExpression(
            dynamic_tokens='p',
            only_first=True,
        ),
        glissando=spannertools.Glissando(),
        staff_lines_spanner=spannertools.StaffLinesSpanner(
            lines=[-4, 4],
            overrides={
                'note_head__style': 'cross',
            }
        ),
        text_spanner=abbreviations.make_text_spanner('inside'),
    ),
    color=None,
    labels=['pedaled'],
)
```
26     pitch_handler=consort.AbsolutePitchHandler(
27         forbid_repetitions=True,
28         pitch_specifier="f c' g' c' f g' c' g' f",
29     ),
30     rhythm_maker=consort.CompositeRhythmMaker(
31         last=rhythmmakertools.IncisedRhythmMaker(
32             inciseSpecifier=rhythmmakertools.InciseSpecifier(
33                 prefix_counts=[0],
34                 suffix_talea=[1],
35                 suffix_counts=[1],
36                 talea_denominator=16,
37             ),
38             duration_spellingSpecifier=rhythmmakertools.DurationSpellingSpecifier(
39                 forbidden_written_duration=(1, 4),
40                 forbid_meter_rewriting=True,
41                 spell_metrically='unassignable',
42             ),
43             tie_specifier=rhythmmakertools.TieSpecifier(
44                 strip_ties=True,
45             ),
46         ),
47         default=rhythmmakertools.EvenDivisionRhythmMaker(  
48             denominators=(4,),
49         ),
50         duration_spellingSpecifier=rhythmmakertools.DurationSpellingSpecifier(
51             forbidden_written_duration=(1, 4),
52             forbid_meter_rewriting=True,
53             spell_metrically='unassignable',
54         ),
55     ),
56 )

D.2.27  ersilia.materials.piano_tremolo_musicSpecifier

```


D.2.28 ersilia.materials.pitch_pipe_musicSpecifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import markuptools
from abjad.tools import rhythmakertools
from abjad.tools import selectortools
from ersilia.materials import abbreviations

pitch_pipe_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector()
                .by_logical_tie()
                .get_slice(start=1, apply_to_each=False)
                [0]
        ),
        dynamic_expressions=consort.DynamicExpression(
            division_period=2,
            dynamic_tokens='p ppp mf',
            start_dynamic_tokens='o fp',
        ),
        inhale_exhale=consort.AttachmentExpression(
            attachments=[
```
```python
markuptools.Markup('exhale', Up)
    .italic().smaller().pad_around(0.5).box(),
markuptools.Markup('inhale', Up)
    .italic().smaller().pad_around(0.5).box(),
markuptools.Markup('inhale', Up)
    .italic().smaller().pad_around(0.5).box(),
    ],
    selector=selectortools.select_pitched_runs()[0],
    ),
    percussion_staff=abbreviations.percussion_staff
    ),
    color='blue',
    labels=['pitch pipes'],
    pitch_handler=consort.AbsolutePitchHandler(  
        pitches_are_nonsemantic=True,
    ),
    rhythm_maker=rhythmmakertools.EvenDivisionRhythmMaker(  
        denominators=[8],
        output_masks=[
            rhythmmakertools.SustainMask(  
                indices=[0, 2],
                period=3,
            ),
            rhythmmakertools.SustainMask(  
                indices=[0, -1],
            ),
        ],
        tieSpecifier=rhythmmakertools.TieSpecifier(  
            tie_across_divisions=True,
        ),
    ),
)

D.2.29 ersilia.materials.saxophone_agitato_musicSpecifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import scoretools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

saxophone_agitato_musicSpecifier = consort.MusicSpecifier(  
    attachment_handler=consort.AttachmentHandler(  
        dynamic_expressions=consort.DynamicExpression(  
            division_period=2,
            dynamic_tokens='mf mp fff',
            start_dynamic_tokens='f',
            stop_dynamic_tokens='mf',
        ),
        slur=consort.AttachmentExpression(  
            attachment_type='slur',
            attachment_style='continuous',
            attachment_cleanup='split',
            attachment_size=0.5,
        ),
    ),
)```
attachments=spannertools.slur(),
selector=selectortools.selector()
    .by_logical_tie(pitched=True)
    .by_duration('==', (1, 16), preprolated=True)
    .by_contiguity()
    .by_length('>', 1)
    .by_pattern(
        pattern=rhythmmakertools.BooleanPattern(
            indices=[0], period=2,
        ),
    )
    .by_leaves()
),

staccati=consort.attachmentexpression(
    attachments=indicatortools.articulation('staccato'),
    selector=selectortools.selector()
        .by_logical_tie(pitched=True)
        .by_duration('==', (1, 16), preprolated=True)
        .by_contiguity()
        .by_length('>', 1)
        .by_pattern(
            pattern=rhythmmakertools.BooleanPattern(
                indices=[1], period=2,
            ),
        )
        .by_leaves()
    [1:]
),

stopped=consort.attachmentexpression(
    attachments=indicatortools.articulation('stopped'),
    selector=selectortools.selector()
        .by_leaves()
        .byRun(scoretools.note)[0]
),

trill_spanner=consort.attachmentexpression(
    attachments=spannertools.complextrillspanner(
        interval='+m3',
    ),
    selector=selectortools.selector()
        .by_logical_tie(pitched=True)
        .by_duration('>', (1, 16), preprolated=True)
        .by_contiguity()
        .by_leaves()
),

color='magenta',
pitch_handler=consort.pitchclasspitchhandler(
    forbid_repetitions=True,
    leap_constraint=12,
    pitchSpecifier=abbreviations.agitato_pitch_specifier,
    registerSpecifier=consort.registerSpecifier(
        base_pitch='C2',
        phrase_inflections=consort.registerinflection.zigzag(6))

937
.reverse() .align(),
    segment_inflections=consort.RegisterInflection.descending(
        width=12) .align()
)

register_spread=6,
)

rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
    extra_counts_per_division=[0, 0, 1, 2, 0, 1],
    output_masks=[
        rhythmmakertools.SustainMask(
            indices=[1],
            period=3,
        ),
    ],

talea=rhythmmakertools.Talea(
    counts=[
        1, -1,
        1, 1, -1,
        1, 1, 1, -1,
        1, 1, 1, 1, 1, -1,
        1,
    ],
    denominator=16,
),
)

D.2.30  ERSILIA.MATERIALS.SHAKER_DECELERANDO_MUSIC_SPECIFIER

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from ersilia.materials import abbreviations

shaker_decelerando_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        staccati=indicatortools.Articulation('staccato'),
        dynamic_expression=consort.DynamicExpression(
            start_dynamic_tokens='f mf mp',
            stop_dynamic_tokens='p',
        ),
        percussion_staff=abbreviations.percussion_staff,
        text_spanner=consort.AttachmentExpression(
            attachments=abbreviations.make_text_spanner('shaker'),
            selector=selectortools.Selector().by_leaves(),
        ),
        color='blue',
        labels=['shakers'],
    ),
    pitch_handler=consort.AbsolutePitchHandler(
        pitches_are_nonsemantic=True,
    ),
)
D.2.31 ersilia.materials.shaker_sporadic_musicSpecifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhytmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

shaker_sporadic_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expression=consort.DynamicExpression(
            dynamic_tokens='ppp',
            transitions=['constante'],
        ),
        percussion_staff=abbreviations.percussion_staff,
        staccati=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('staccato'),
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('<=', (1, 16), preprolated=True)
            [0],
        ),
        text_spanner=consort.AttachmentExpression(
            attachments=abbreviations.make_text_spanner('shaker'),
            selector=selectortools.Selector().by_leaves(),
        ),
        tremolo=consort.AttachmentExpression(
            attachments=spannertools.StemTremoloSpanner(),
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('>', (1, 16), preprolated=True)
        ),
    ),
    rhythmMaker=rhythmmakertools.AccelerandoRhythmMaker(
        beamSpecifier=rhythmmakertools.BeamSpecifier(
            use_feather_beams=True,
        ),
        duration_specification=rhythmmakertools.DurationSpellingSpecifier(
            forbid_meter_rewriting=True,
        ),
        interpolation_specifiers=rhythmmakertools.InterpolationSpecifier(
            start_duration=durationtools.Duration(1, 32),
            stop_duration=durationtools.Duration(1, 8),
            written_duration=durationtools.Duration(1, 32),
        ),
        tuplet_specifiers=rhythmmakertools.TupletSpellingSpecifier(
            use_note_duration_bracket=True,
        ),
    ),
)
```
D.2.32  ersilia.materials.shaker_tremolo_musicSpecifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import scoretools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

shaker_tremolo_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector().by_leaves()
            .by_run(scoretools.Note)
            .by_leaves()
        ).by_leaves()
        .by_run(scoretools.Note)
        [0],
    ),
    dynamic_expression=consort.DynamicExpression(
        dynamic_tokens='fp',
    ),
    percussion_staff=abbreviations.percussion_staff,
    text_spanner=consort.AttachmentExpression(
        attachments=abbreviations.make_text_spanner('shaker'),
        selector=selectortools.Selector().by_leaves()
    ),
    tremolo_spanner=spannertools.StemTremoloSpanner(),
    color='blue',
    labels=['shakers'],
```
pitch_handler=consort.AbsolutePitchHandler(
    pitches_are_nonsemantic=True,
),

rhythm_maker=rhythmmakertools.NoteRhythmMaker(
    tieSpecifier=rhythmmakertools.TieSpecifier(
        tie_across_divisions=True,
    ),
),

D.2.33  
ERSILIA.MATERIALS.SPARSE_TIMESPAN MAKER

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import durationtools
from abjad.tools import rhythmmakertools

sparse_timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=rhythmmakertools.Talea(
        counts=[1, 0, 3, 2, 4, 5, 1, 2],
        denominator=8,
    ),
    playing_talea=rhythmmakertools.Talea(
        counts=[2, 3, 2, 2, 3],
        denominator=8,
    ),
    playing_groupings=[1],
    repeat=True,
    silence_talea=rhythmmakertools.Talea(
        counts=[4, 8, 6],
        denominator=8,
    ),
    step_anchor=Right,
    synchronize_groupings=False,
    synchronize_step=False,
    timespanSpecifier=consort.TimespanSpecifier(
        minimum_duration=durationtools.Duration(1, 8),
    ),
)

D.2.34  
ERSILIA.MATERIALS.STRING_AGITATO_MUSICSpecifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

string_agitato_musicSpecifier = consort.MusicSpecifier(
    attachmentHandler=consort.AttachmentHandler(

941
dynamic_expressions=consort.DynamicExpression(
    division_period=2,
    dynamic_tokens='mp fff',
    start_dynamic_tokens='f p',
    stop_dynamic_tokens='p f',
),
harmonics=consort.AttachmentExpression(
    attachments=consort.HarmonicExpression('P4'),
    is-destructive=True,
    selector=selectortools.Selector()
      .by_logical_tie(pitched=True)
      .by_duration('==', (1, 16), preprolated=True)
      .by_pattern(
        pattern=rhythmmakertools.BooleanPattern(
          indices=[2, 4],
          period=5,
        ),
      ),
),
slur=consort.AttachmentExpression(
    attachments=spannertools.Slur(),
    selector=selectortools.Selector()
      .by_logical_tie(pitched=True)
      .by_duration('==', (1, 16), preprolated=True)
      .by_contiguity()
      .by_length('>', 1)
      .by_leaves()
),
staccati=consort.AttachmentExpression(
    attachments=indicatortools.Articulation('staccato'),
    selector=selectortools.Selector()
      .by_logical_tie(pitched=True)
      .by_duration('==', (1, 16), preprolated=True)
      .by_contiguity()
      .by_leaves()
),
accents=consort.AttachmentExpression(
    attachments=indicatortools.Articulation('accent'),
    selector=selectortools.Selector()
      .by_logical_tie(pitched=True)
      .by_duration('>', (1, 16), preprolated=True)
      .by_contiguity()
      .by_leaves()
      [0]
),
trill_spanner=consort.AttachmentExpression(
    attachments=[
      spannertools.ComplexTrillSpanner('+m3'),
      spannertools.ComplexTrillSpanner('+P4'),
    ],
    selector=selectortools.Selector()
      .by_logical_tie(pitched=True)
      .by_duration('>', (1, 16), preprolated=True)
      .by_contiguity()
.by_leaves()
),
),
color='magenta',
labels=[],
pitch_handler=consort.PitchClassPitchHandler(
    forbid_repetitions=True,
pitch_application_rate='division',
pitchSpecifier=abbreviations.agitato_pitchSpecifier,
registerSpecifier=consort.RegisterSpecifier(
    base_pitch='G3',
    phrase_inflections=consort.RegisterInflection.zigzag(6)
    .reverse()
    .align(),
    segment_inflections=consort.RegisterInflection
    .descending(width=6)
    .align()
    .reverse()
    .align(),
    register_spread=3,
),
),
rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
    extra_counts_per_division=[0, 0, 1, 2, 0, 1],
    output_masks=[
        rhythmmakertools.SustainMask(
            indices=[1],
            period=3,
        ),
    ],
    talea=rhythmmakertools.Talea(
        counts=[
            1, -1,
            1, 1, -1,
            1, 1, 1, -1,
            1, 1, 1, 1, -1,
            1, 1, 1, 1, 1, -1,
            1, 1, 1, 1, -2,
        ],
        denominator=16,
    ),
),
)

D.2.35 ersilia.materials.string_legato_music_specifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import durationtools
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools

string_legato_musicSpecifier = consort.MusicSpecifier(
attachment_handler=consort.AttachmentHandler(
    dynamic_expressions=consort.DynamicExpression(
        dynamic_tokens='p mf',
        start_dynamic_tokens='p fp',
        stop_dynamic_tokens='o ff',
    ),
    glissando=spannertools.Glissando(),
    tenuti=consort.AttachmentExpression(
        attachments=indicatortools.Articulation('tenuto'),
        selector=selectortools.Selector()
        .by_leaves()
        .by_logical_tie(pitched=True)
        .by_pattern(
            rhythmmakertools.BooleanPattern(
                indices=[3],
                period=4,
            ),
            [0]
        ),
    ),
    tremolo_trill=consort.AttachmentExpression(
        attachments=(
            spannertools.ComplexTrillSpanner(interval='+m3'),
            spannertools.StemTremoloSpanner(),
            spannertools.ComplexTrillSpanner(interval='+m3'),
            spannertools.ComplexTrillSpanner(interval='+M2'),
            spannertools.StemTremoloSpanner(),
        ),
        selector=selectortools.Selector()
        .by_leaves()
        .by_logical_tie(pitched=True)
        .by_pattern(
            rhythmmakertools.BooleanPattern(
                indices=[0, 1, 2],
                period=4,
            ),
        ),
        color='green',
        labels=[],
    ),
    pitch_handler=consort.AbsolutePitchHandler(
        pitchSpecifier="d' f' d' fqs' ef' d' ef' f' fqs' d' g' d' d' as'", 
        pitch_application_rate='division',
    ),
    rhythm_maker=rhythmmakertools.EvenDivisionRhythmMaker(
        denominators=[8, 4, 8, 1],
        duration_spellingSpecifier=rhythmmakertools.DurationSpellingSpecifier(
            forbidden_written_duration=durationtools.Duration(1, 2),
        ),
        extra_counts_per_division=[0, 1, 0, 2, 1],
    )
)
from abjad.tools import indicatortools
from abjad.tools import selectortools
from abjad.tools import spannertools

d = string_low_pedal_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector()
                .by_logical_tie()
                .get_slice(start=1, apply_to_each=False)
                [0]
        ),
        dynamic_expressions=consort.DynamicExpression(
            division_period=2,
            dynamic_tokens='p ppp',
            start_dynamic_tokens='o',
            stop_dynamic_tokens='o',
        ),
        glissando=spannertools.Glissando(),
    ),
    color=None,
    labels=[],
    minimum_phrase_duration=(3, 2),
    pitch_handler=consort.PitchClassPitchHandler(
        pitch_application_rate='phrase',
        deviations=[0, 0, 0, 0.5],
        pitch_specifier='d f d g f d f',
        registerSpecifier=consort.RegisterSpecifier(
            base_pitch='C4',
        ),
        register_spread=0,
    ),
    rhythm_maker=rhythmmakertools.EvenDivisionRhythmMaker(
        denominators=[8],
        output_masks=[
            rhythmmakertools.SustainMask(
                indices=[0, 1],
                period=3,
            ),
            rhythmmakertools.SustainMask(
                indices=[0, -1],
            ),
        ],
        tieSpecifier=rhythmmakertools.TieSpecifier(
            tie_across_divisions=True,
        ),
    ),
)
```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

string_ostinato_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expressions=consort.DynamicExpression(
            dynamic_tokens='p',
        ),
        pizzicati=consort.AttachmentExpression(
            attachments=[
                abbreviations.make_text_markup('pizz.'),
                indicatortools.Articulation('snappizzicato'),
            ],
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_contiguity()
                .by_length('==', 1)
                .by_duration('==', (1, 16), preprolated=True)
                .by_leaves()
            [0]
        ),
        tenuti=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('tenuto'),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_contiguity()
                .by_length('==', 1)
                .by_duration('>', (1, 16), preprolated=True)
                .by_leaves()
            [0]
        ),
        slur=consort.AttachmentExpression(
            attachments=spannertools.Slur(),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_contiguity()
                .by_length('>', 1)
                .by_leaves()
            ),
        color='darkyellow',
    pitch_handler=consort.PitchClassPitchHandler(
```
```
deviations=[0, 0, 0, 0.5, 0, -0.5],
forbid_repetitions=True,
leap_constraint=6,
pitch_specifier='d f d f d f c f bf d f df',
register_specifier=consort.RegisterSpecifier(
    base_pitch='C4',
    segment_inflections=consort.RegisterInflection
    .zigzag(6)
    .reverse(),
),
register_spread=3,
)

rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
    extra_counts_per_division=[0, 0, 1, 2, 0, 1],
talea=rhythmmakertools.Talea(
    counts=[1, 1, -3, 2, 1, -2, 3, 1, -3],
denominator=16,
)
)

D.2.38 ersilia.materials.string_overpressure_musicSpecifier

string_overpressure_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            [0]
        ),
    ),
dynamic_expressions=consort.DynamicExpression(
    start_dynamic_tokens='fff',
    only_first=True,
),
text_spanner=consort.AttachmentExpression(
    attachments=abbreviations.make_text_spanner('overpressure'),
    selector=selectortools.Selector().by_leaves(),
),
),
color=None,
labels=[],
pitch_handler=consort.AbsolutePitchHandler(
    deviations=[0, 1, 0, 0.5],
    logical_tie_expressions=[
        consort.ChordExpression(chord_expr=[0, 7]),
    ],
)
D.2.39 ersilia.materials.string_pointillist_music_specifier

string_pointillist_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        mordents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('mordent'),
            selector=selectortools.Selector()
        ),
        dynamic_expressions=consort.DynamicExpression(
            start_dynamic_tokens='ppp',
            only_first=True,
        ),
        text_spanner=consort.AttachmentExpression(
            attachments=consort.AttachmentExpression(
                attachments=abbreviations.make_text_spanner('pizz.'),
                selector=selectortools.Selector() by_leaves()
            ),
            selector=selectortools.Selector() by_leaves()
        ),
        color='darkyellow',
        labels=[],
    ),
)
.pitch_handler = consort.PitchClassPitchHandler(
    forbid_repetitions=True,
    pitch_specifier=abbreviations.agitato_pitch_specifier,
    register_specifier=consort.RegisterSpecifier(
        base_pitch='G3',
        phrase_inflections=consort.RegisterInflection
        .zigzag()
        .reverse()
        .align(),
        segment_inflections=consort.RegisterInflection
        .descending(width=3)
        .align(),
    ),
),

rhythm_maker = consort.CompositeRhythmMaker(
    default=rhythmmakertools.TaleaRhythmMaker(
        extra_counts_per_division=[0, 0, 1],
        talea=rhythmmakertools.Talea(
            counts=[1, -1, 1, -2, 1, -3],
            denominator=16,
        ),
    ),
    last=rhythmmakertools.IncisedRhythmMaker(
        inciseSpecifier=rhythmmakertools.InciseSpecifier(
            fill_with_notes=False,
            prefix_counts=[1],
            prefix_talea=[1],
            talea_denominator=16,
        ),
    ),
)

D.2.40 ersilia.materials.string_tremolo_music_specifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

string_tremolo_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector()
            .by_logical_tie()
            .by_duration('==', (1, 16)),
            preprolated=True
            .with_next_leaf()
        ),
        dynamic_expressions=consort.DynamicExpression
    )
)
division_period=2,
dynamic_tokens='p ppp mp',
start_dynamic_tokens='o fp',
),
stem_tremolo=spannertools.StemTremoloSpanner(),
),
color='red',
labels=[],
pitch_handler=consort.PitchClassPitchHandler(
    deviations=[0, 0.5, 0, -0.5],
    logical_tie_expressions=[
        consort.ChordExpression([0, 8]),
    ],
pitch_application_rate='phrase',
pitchSpecifier=abbreviations.agitato_pitchSpecifier,
pitch_operationSpecifier=abbreviations.pitch_operationSpecifier,
registerSpecifier=consort.RegisterSpecifier(
    base_pitch='G3',
),
),
rhythm_maker=rhythmmakertools.EvenDivisionRhythmMaker(
    denominators=[16],
    extra_counts_per_division=[0, 1],
    output_masks=[
        rhythmmakertools.SustainMask(
            indices=[2],
            period=3,
        ),
        rhythmmakertools.SustainMask(
            indices=[0, -1],
        ),
    ],
tieSpecifier=rhythmmakertools.TieSpecifier(
    tie_across_divisions=True,
),
),
)

D.2.41 ersilia.materials.sustained_timespan_maker

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import durationtools
from abjad.tools import rhythmmakertools

sustained_timespan_maker = consort.TaleaTimespanMaker(
    initial_silence_talea=rhythmmakertools.Talea(
        counts=(0, 2, 1),
        denominator=8,
    ),
    playing_talea=rhythmmakertools.Talea(
        counts=(4, 5, 4, 3, 7, 6),
        denominator=8,
D.2.42 ersilia.materials.tutti_timespan_maker

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import durationtools
from abjad.tools import rhythmikertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

tutti_timespan_maker = consort.TaleaTimespanMaker(
 fuse_groups=True,
 playing_talea=rhythmikertools.Talea(
    counts=(4, 5, 4, 3, 7, 6),
    denominator=8,
    ),
 playing_groupings=(3, 4, 2, 3, 5),
 repeat=True,
 silence_talea=rhythmikertools.Talea(
    counts=(3, 4, 2, 5, 6, 9),
    denominator=4,
    ),
 step_anchor=Right,
 synchronize_groupings=True,
 synchronize_step=True,
 timespanSpecifier=consort.TimespanSpecifier(
    minimum_duration=durationtools.Duration(1, 4),
    ),
)
```

D.2.43 ersilia.materials.wind_agitato_musicSpecifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmikertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations
```
wind_agitato_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        accents_short=consort.AttachmentExpression(
            attachments=[
                indicatortools.Articulation('accent'),
                indicatortools.Articulation('staccatissimo'),
            ],
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                .by_duration('==', (1, 16), preprolated=True)
                .by_contiguity()
                .by_length('==', 1)
                .by_leaves()
                [0]
        ),
        accents_long=indicatortools.Articulation('accent'),
        selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('>', (1, 16), preprolated=True)
            .by_contiguity()
            .by_length('==', 1)
            .by_leaves()
            [0]
    ),
    dynamic_expressions=consort.DynamicExpression(
        division_period=2,
        dynamic_tokens='mf mp fff',
        start_dynamic_tokens='f',
        stop_dynamic_tokens='p mp mf',
    ),
    flutter_tongue=indicatortools.AttachmentExpression(
        attachments=abbreviations.make_text_spanner('Flz.'),
        selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('>', (1, 16), preprolated=True)
            .by_contiguity()
            .by_leaves()
    ),
    slur=consort.AttachmentExpression(
        attachments=spannertools.Slur(),
        selector=selectortools.Selector()
            .by_logical_tie(pitched=True)
            .by_duration('==', (1, 16), preprolated=True)
            .by_contiguity()
            .by_length('>', 1)
            .by_pattern(
                pattern=rhythmmakertools.BooleanPattern(
                    indices=[0],
                    period=2,
                ),
            )
    )
.by_leaves()
),

staccato=consort.AttachmentExpression(
    attachments=indicatorools.Articulation('staccato'),
    selector=selectortools.Selector()
       .by_logical_tie(pitched=True)
       .by_duration('==', (1, 16), preprolated=True)
       .by_contiguity()
       .by_length('>', 1)
       .by_pattern(
          pattern=rhythmmakertools.BooleanPattern(
              indices=[0, 2],
              period=3,
          ),
       )
    )
    .by_leaves()
),

stem_tremolo_spanner=consort.AttachmentExpression(
    attachments=spannertools.StemTremoloSpanner(),
    selector=selectortools.Selector()
       .by_logical_tie(pitched=True)
       .by_duration('>', (1, 16), preprolated=True)
       .by_contiguity()
       .by_leaves()
    ),

text color='magenta',
labels=[]
pitch_handler=consort.PitchClassPitchHandler(
    forbid_repetitions=True,
    pitch_specifier=abbreviations.agitato_pitch_specifier,
    registerSpecifier=consort.RangeSpecifier(  
        base_pitch='C4',
        phrase_inflections=consort.RegisterInflection.zigzag(6)
    ).reverse()
    .align(),
    segment_inflections=consort.RegisterInflection
        .descending(width=12)
    .align()
    ),
    register_spread=6,
),
rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
    extra_counts_per_division=[0, 0, 1, 2, 0, 1],
    output_masks=[
        rhythmmakertools.SustainMask(
            indices=[1],
            period=3,
        ),
    ],
    talea=rhythmmakertools.Talea(
        counts=[
            1, -1,
            1, 1, -1,
        ],
    )
D.2.44 ersilia.materials.wind_continuo_musicSpecifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmakertools
from abjad.tools import scoretools
from abjad.tools import selectortools

wind_continuo_musicSpecifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expressions=consort.DynamicExpression(
            division_period=2,
            dynamic_tokens='p mp mf',
            start_dynamic_tokens='o',
            stop_dynamic_tokens='o',
        ),
        staccati=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('staccato'),
            selector=selectortools.select_pitched_runs(),
        ),
        color=None,
        labels=[],
    ),
    pitch_handler=consort.PitchClassPitchHandler(
        deviations=[0, 2, 0, 3, 0, 3, 0, 2, 0, 5, 0, 3, 0, 5, 0, 8, 7],
        pitch_specifier="d' f' df'",
        pitch_application_rate='division',
        registerSpecifier=consort.RegisterSpecifier(
            base.pitch='C4',
        ),
    ),
    rhythm_maker=rhythmakertools.EvenDivisionRhythmMaker(
        burnishSpecifier=rhythmakertools.BurnishSpecifier(
            left_classes=[scoretools.Rest],
            left_counts=[1, 1, 0, 0, 1, 0],
            right_classes=[scoretools.Rest],
            right_counts=[1, 0],
        ),
        denominators=[16],
        extra_counts_per_division=(0, 0, 1, 2, 0, 1),
    )
)```

954
D.2.45  ersilia.materials.wind_low_pedal_music_specifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import rhythmmakertools

wind_low_pedal_music_specifier = consort.
MusicSpecifier(
    attachment_handler=consort.
    AttachmentHandler(
        dynamic_expressions=consort.
        DynamicExpression(
            division_period=2,
            dynamic_tokens='p ppp',
            start_dynamic_tokens='o',
            stop_dynamic_tokens='o',
        ),
    ),
    color=None,
    labels=[],
    minimum_phrase_duration=(3, 2),
    pitch_handler=consort.
    AbsolutePitchHandler(
        pitch_specifier='D2 F2 D2 G2 F2 D2 F2',
    ),
    rhythm_maker=rhythmmakertools.
    NoteRhythmMaker(
        tie_specifier=rhythmmakertools.
        TieSpecifier(
            tie_across_divisions=True,
        ),
    ),
)
```

D.2.46  ersilia.materials.wind_ostinato_music_specifier

```python
# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools

wind_ostinato_music_specifier = consort.
MusicSpecifier(
    attachment_handler=consort.
    AttachmentHandler(
        dynamic_expressions=consort.
        DynamicExpression(
            dynamic_tokens='p',
        ),
    ),
    color=consort.
    AttachmentExpression(
        attachments=spannertools.
        Slur(),
    ),
    slur=consort.
    AttachmentExpression(
        selector=selectortools.
        Selector()
        .by_logical_tie(pitched=True)
        .by_duration('==', (1, 16), preprolated=True)
        .by_contiguity()
        .by_length('>', 1)
        .by_leaves()
    ),
    staccati=consort.
    AttachmentExpression(
```
24    attachments=indicatortools.Articulation('staccato'),
25    selector=selectortools.Selector()
26        .by_logical_tie(pitched=True)
27        .by_duration('==', (1, 16), preprolated=True)
28        .by_contiguity()
29        .by_leaves()
30        [-1]
31    ),
32    color='darkyellow',
33    pitch_handler=consort.AbsolutePitchHandler(
34        forbid_repetitions=True,
35        pitch_specifier="d' f''",
36    ),
37    rhythm_maker=rhythmmakertools.TaleaRhythmMaker(
38        extra_counts_per_division=[0, 0, 1, 2, 0, 1],
39        talea=rhythmmakertools.Talea(
40            counts=[1, 1, -3],
41            denominator=16,
42        ),
43    ),
44)
45}

D.2.47 ersilia.materials.wind_pointillist_music_specifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from ersilia.materials import abbreviations

wind_pointillist_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        dynamic_expressions=consort.DynamicExpression(
            start_dynamic_tokens='ppp',
            only_first=True,
        ),
        mordent=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('mordent'),
            selector=selectortools.Selector()
                .by_logical_tie(pitched=True)
                [0]
        ),
    ),
    color='darkyellow',
    labels=[],
    pitch_handler=consort.PitchClassPitchHandler(
        forbid_repetitions=True,
        pitch_specifier=abbreviations.agitato_pitch_specifier,
        register_specifier=consort.RegisterSpecifier(
            base_pitch='C4',
            phrase_inflections=consort.RegisterInflection
30),
31    ),
32    color='darkyellow',
33    pitch_handler=consort.AbsolutePitchHandler(
34        forbid_repetitions=True,
35        pitch_specifier=abbreviations.agitato_pitch_specifier,
36        register_specifier=consort.RegisterSpecifier(
37            base_pitch='C4',
38            phrase_inflections=consort.RegisterInflection
39    ),
40)
.zigzag(6)
.reverse()
.align(),
    segment_inflections=consort.RegisterInflection
    .descending(width=6)
    .align()
),
    rhythm_maker=consort.CompositeRhythmMaker(
        default=rhythmmakertools.TaleaRhythmMaker(
            extra_counts_per_division=[0, 1],
            talea=rhythmmakertools.Talea(
                counts=[1, 1, -1, 1, -2, 1, 1, -3],
                denominator=16,
            ),
        ),
    ),
    last=rhythmmakertools.IncisedRhythmMaker(
        incise_specifier=rhythmmakertools.InciseSpecifier(
            fill_with_notes=False,
            prefix_counts=[2],
            prefix_talea=[1, 0],
            talea_denominator=16,
        ),
    ),
)

D.2.48 ersilia.materials.wind_tremolo_music_specifier

# -*- encoding: utf-8 -*-
import consort
from abjad.tools import indicatortools
from abjad.tools import rhythmmakertools
from abjad.tools import selectortools
from abjad.tools import spannertools
from ersilia.materials import abbreviations

wind_tremolo_music_specifier = consort.MusicSpecifier(
    attachment_handler=consort.AttachmentHandler(
        outer_accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.select_pitched_runs()[0],
        ),
        inner_accents=consort.AttachmentExpression(
            attachments=indicatortools.Articulation('accent'),
            selector=selectortools.Selector()
            .by_logical_tie()
            .by_duration('==', (1, 8), preprolated=True)
            .with_next_leaf()
        ),
        dynamic_expressions=consort.DynamicExpression(
            division_period=2,
            dynamic_tokens='p ppp mp',
    )
}
```python
start_dynamic_tokens='fp',
stop_dynamic_tokens='mf ppp',
trill_spanner=consort.AttachmentExpression(
    attachments=spannertools.ComplexTrillSpanner('M2'),
    selector=selectortools.select_pitched_runs(),
),
color='red',
labels=[],
pitch_handler=consort.PitchClassPitchHandler(
    deviations=[0, 1],
pitch_application_rate='phrase',
pitch_specifier=abbreviations.agitato_pitch_specifier,
pitch_operation_specifier=abbreviations.pitch_operation_specifier,
register_specifier=consort.RegisterSpecifier(
    base_pitch='C4',
),
),
rhythm_maker=rhythmmakertools.EvenDivisionRhythmMaker(
    denominators=[8],
    extra_counts_per_division=[0, 1, 2],
    output_masks=[
        rhythmmakertools.SustainMask(
            indices=[2],
            period=3,
        ),
        rhythmmakertools.SustainMask(
            indices=[0, -1],
        ),
    ],
tie_specifier=rhythmmakertools.TieSpecifier(
    tie_across_divisions=True,
),
),
```

D.3 **ERSILLA**

D.3.1 **ERSILLA.SEGMENTS.CHEMISH**

```python
### SEGMENT ###
 segment_maker = ersilia.ErsiliaSegmentMaker(
    ...)
```
desired_duration_in_seconds=90,
name='Chemish',
permitted_time_signatures=ersilia.permitted_time_signatures,
tempo=abjad.Tempo((1, 4), 80),
)

### PEDAL ###
segment_maker.add_setting(
timespan_identifier=timespantools.Timespan(start_offset=1),
timespan_maker=new(ersilia.sustained_timespan_maker,
fuse_groups=True,
),
percussion=ersilia.percussion_low_pedal_musicSpecifier,
)

### TREMOLO ###
segment_maker.add_setting(
timespan_identifier=[-1, 1],
timespan_maker=new(ersilia.sustained_timespan_maker,
fuse_groups=True,
),
bass=new(ersilia.string_low_pedal_musicSpecifier,
pitch_handler__register_spread=0,
).transpose('E1'),
)

segment_maker.add_setting(
timespan_identifier=[-1, 2,
-3, 10,
-4,
],
timespan_maker=new(ersilia.sustained_timespan_maker,
fuse_groups=True,
),
flute=ersilia.wind_tremolo_musicSpecifier,
clarinet=ersilia.wind_tremolo_musicSpecifier,
)

segment_maker.add_setting(
timespan_maker=new(ersilia.sparse_timespan_maker,
silence_talea__denominator=4,
),
violin=ersilia.string_tremolo_musicSpecifier,
viola=new(ersilia.string_tremolo_musicSpecifier,
pitch_handler__registerSpecifier__base_pitch='C3',
)
célo = new(ersilia.string_tremolo_music_specifier,
pitch_handler__register_specifier__base_pitch='C2'),
)

### CONTINUO ###
segment_maker.add_setting(
timespan_identifier=[-1, 1, -2, 1, -3, 1, -2],
timespan_maker=ersilia.sparse_timespan_maker,
piano_lh=ersilia.piano_string_glissando_music_specifier,
)

### OSTINATO ###
### AGITATO ###
music_specifier = new(ersilia.saxophone_agitato_music_specifier,
attachment_handler__dynamic_expressions=consort.DynamicExpression(
  start_dynamic_tokens='fp o',
  stop_dynamic_tokens='mp mf p f o',
  dynamic_tokens='pp p',
),
pitch_handler__register_specifier__base_pitch='C4',
pitch_handler__register_specifier__segment_inflections=consort.RegisterInflection
  .ascending(width=12)
  .align()
)
segment_maker.add_setting(
timespan_identifier=[-1, 6, 
  -5, 1, 
  -1, 2, 
  -1, 1, 
],
timespan_maker=ersilia.dense_timespan_maker,
saxophone=music_specifier,
)
segment_maker.add_setting(
timespan_maker=ersilia.sparse_timespan_maker,
flute=music_specifier,
oboe=music_specifier,
clarinet=music_specifier.transpose(-12),
)
segment_maker.add_setting(
timespan_identifier=[-1, 5],
timespan_maker=ersilia.sparse_timespan_maker,
guitar=ersilia.guitar_strummed_music_specifier,
### POINTILLIST ###

```python
segment_maker.add_setting(
    timespan_identifier=[-1, 1, -1, 1, -1],
    timespan_maker=ersilia.sparse_timespan_maker,
    piano_rh=new(ersilia.piano_pointillist_musicSpecifier,
        pitch_handler__leap_constraint=8,
    ),
)
```

### AUXILIARY ###

```python
segment_maker.add_setting(
    timespan_identifier=[4, -1, 2, -1, 1, -1, 5, 3],
    timespan_maker=new(ersilia.sustained_timespan_maker,
        fuse_groups=True,
        initial_silence_talea=None,
        padding=(1, 4),
        playing_groupings=[2, 3, 2, 4, 6],
        silence_talea__denominator=2,
        timespanSpecifier=consort.TimespanSpecifier(
            minimum_duration=0,
        ),
    ),
    flute=ersilia.shaker_tremolo_music_specifier,
    clarinet=ersilia.shaker_tremolo_music_specifier,
    oboe=ersilia.shaker_tremolo_music_specifier,
    violin=ersilia.shaker_tremolo_music_specifier,
    viola=ersilia.shaker_tremolo_music_specifier,
    cello=ersilia.shaker_tremolo_music_specifier,
)
```

```python
#segment_maker.add_setting(  
#    timespan_identifier=[-2, 1, -3, 1, -5, 1, -3],  
#    timespan_maker=new(  
#        ersilia.sparse_timespan_maker,  
#        padding=(1, 4),  
#        timespanSpecifier=consort.TimespanSpecifier(  
#            minimum_duration=0,  
#        ),  
#    ),  
#    flute=ersilia.shaker_decelerando_music_specifier,  
#    clarinet=ersilia.shaker_decelerando_music_specifier,  
#    oboe=ersilia.shaker_decelerando_music_specifier,  
#    violin=ersilia.shaker_decelerando_music_specifier,  
#    viola=ersilia.shaker_decelerando_music_specifier,  
#    cello=ersilia.shaker_decelerando_music_specifier,  
#)
```
# viola=ersilia.shaker_decelerando_musicSpecifier,
# cello=ersilia.shaker_decelerando_musicSpecifier,
#)

segment_maker.add_setting(
    timespan_identifier=[
        -4, 1,
        -4, 1,
        -4, 1,
        -4, 1,
        -2, 2,
    ],
    timespan_maker=new(
        ersilia.dense_timespan_maker,
        fuse_groups=True,
        silence_talea__denominator=4,
        timespan_specifier=consort.TimespanSpecifier(
            minimum_duration=0,
        ),
    ),
    guitar_pp=ersilia.pitch_pipe_musicSpecifier,
    piano_pp=ersilia.pitch_pipe_musicSpecifier,
    percussion_pp=ersilia.pitch_pipe_musicSpecifier,
    bass_pp=ersilia.pitch_pipe_musicSpecifier,
)

### INTERRUPT ###

segment_maker.add_setting(
    timespan_identifier=[-1, 1, -1, 1],
    timespan_maker=consort.BoundaryTimespanMaker(
        labels=['pitch pipes'],
        stop_talea=rhythmmakertools.Talea(
            counts=[2, 9, 3, 1],
            denominator=4,
        ),
    ),
    percussion=new(
        ersilia.percussion_snare_interruption_musicSpecifier,
        rhythm_maker__first__inciseSpecifier__prefix_talea=[1],
        rhythm_maker__first__inciseSpecifier__prefix_counts=[1],
    ),
    silenced_contexts=segment_maker.score_template.all_voice_names,
)

segment_maker.add_setting(
    timespan_identifier=timespantools.Timespan(0, (1, 4)),
    piano_rh=new(
        ersilia.piano_arm_cluster_musicSpecifier,
        attachment_handler__laissez_vibrer=abbreviations.laissez_vibrer,
    ).transpose(12),
)
# --^-- encoding: utf-8 --^--
import ersilia
from abjad.tools import timespantools

### SEGMENT ###
segment_maker = ersilia.ErsiliaSegmentMaker(
    desired_duration_in_seconds=6,
    name='[i]',
    permitted_time_signatures=ersilia.permitted_time_signatures,
    repeat=True,
)

### PEDAL ###

### TREMOLO ###

### CONTINUO ###

### OSTINATO ###

### AGITATO ###
segment_maker.add_setting(
    timespan_identifier=[1, -3],
    timespan_maker=ersilia.dense_timespan_maker,
    flute=ersilia.wind_agitato_musicSpecifier
        .rotate(4),
    clarinet=ersilia.wind_agitato_musicSpecifier
        .rotate(5),
    oboe=ersilia.wind_agitato_musicSpecifier,
    piano_rh=ersilia.piano_agitato_musicSpecifier.rotate(-1),
    piano_lh=ersilia.piano_agitato_musicSpecifier
        .transpose(-24),
    violin=ersilia.string_agitato_musicSpecifier
        .rotate(4),
    viola=ersilia.string_agitato_musicSpecifier
        .transpose('C3')
        .rotate(5),
    cello=ersilia.string_agitato_musicSpecifier
        .transpose('C2')
        .rotate(6),
    bass=ersilia.string_agitato_musicSpecifier
        .transpose('E1')
        .rotate(7),
)

segment_maker.add_setting(
    saxophone=ersilia.saxophone_agitato_musicSpecifier
        .transpose('C2')
)
rotate(6),
guitar=ersilia.guitar_agitato_musicSpecifier
.rotate(),
percussion=ersilia.percussion_tom_fanfare_musicSpecifier,
)

### POINTILLIST ###

### INTERRUPT ###

### AUXILIARY ###

segment_maker.add_setting(
    timespan_identifier=timespanTools.TimeSpan(0, (1, 4)),
    piano_lh=ersilia.piano_arm_cluster_musicSpecifier
    .transpose(-12),
)

### SEGMENT ###

### PEDAL ###

### TREMOLO ###

### CONTINUO ###

### OSTINATO ###

### AGITATO ###

segment_maker.add_setting(
    percussion=ersilia.percussion_tom_fanfare_musicSpecifier,
)

D.3.3 ersilia.segments.cut_2

# -*- encoding: utf-8 -*-
import ersilia
from abjad.tools import timespantools

### SEGMENT ###

segment_maker = ersilia.ErsiliaSegmentMaker(
    desired_duration_in_seconds=6,
    name='[ii]',
    permitted_time_signatures=ersilia.permitted_time_signatures,
    repeat=True,
)

### PEDAL ###

### TREMOLO ###

### CONTINUO ###

### OSTINATO ###

### AGITATO ###

segment_maker.add_setting(
    percussion=ersilia.percussion_tom_fanfare_musicSpecifier,
)

segment_maker.add_setting(
    timespan_identifier=[1, -1],
    timespan_maker=ersilia.dense_timespan_maker,
    flute=[
        ersilia.wind_pointillist_musicSpecifier.rotate(7),
        ersilia.wind_agitato_musicSpecifier.rotate(7),
    ]
)
35  
36  clarinet=[
37     ersilia.wind_agitato_music_specifier.transpose('C2').rotate(8),
38     ersilia.wind_pointillist_music_specifier.transpose('C2').rotate(8),
39  ],
40  oboe=[
41     ersilia.wind_pointillist_music_specifier.rotate(0),
42     ersilia.wind_agitato_music_specifier.rotate(0),
43  ],
44  violin=ersilia.string_agitato_music_specifier
45     .rotate(8),
46  viola=ersilia.string_agitato_music_specifier
47     .transpose('C3')
48     .rotate(3),
49  cello=ersilia.string_agitato_music_specifier
50     .transpose('C2')
51     .rotate(10),
52  bass=ersilia.string_pointillist_music_specifier
53     .transpose('E1')
54     .rotate(11),
55  )
56
57  segment_maker.add_setting(
58      timespan_maker=ersilia.dense_timespan_maker,
59      saxophone=ersilia.saxophone_agitato_music_specifier
60      .transpose('C2'),
61      guitar=ersilia.guitar_agitato_music_specifier
62      .rotate(2),
63      piano_rh=ersilia.piano_palm_cluster_music_specifier,
64      piano_lh=ersilia.piano_agitato_music_specifier
65      .transpose(-24),
66  )
67
68  ### POINTILLIST ###
69
70  ### INTERRUPT ###
71
72  ### AUXILIARY ###

D.3.4 ersilia.segments.komokome

```python
# -*- encoding: utf-8 -*-
import abjad
import consort
import ersilia
from abjad import new
from abjad.tools import rhythmmakertools

segment_maker = ersilia.ErsiliaSegmentMaker(
    desired_duration_in_seconds=120,
    name='Komokome',
    permitted_time_signatures=ersilia.permitted_time_signatures,
    tempo=abjad.Tempo((1, 4), 96),
```

965
### PEDAL ###

segment_maker.add_setting(
    timespan_identifier=[
        1,
        -1, 1,
        -2, 1,
    ],
    clarinet=ersilia.wind_low_pedal_musicSpecifier,
    saxophone=ersilia.wind_low_pedal_musicSpecifier,
    percussion=ersilia.percussion_low_pedal_musicSpecifier,
    bass=ersilia.string_low_pedal_musicSpecifier
    .transpose('E1'),
)

### TREMOLO ###

segment_maker.add_setting(
    timespan_identifier=[
        -1, 2,
        -3, 10,
        -4,
    ],
    timespan_maker=new(
        ersilia.sustained_timespan_maker,
        fuse_groups=True,
    ),
    flute=ersilia.wind_tremolo_musicSpecifier,
)

segment_maker.add_setting(
    timespan_maker=new(
        ersilia.sustained_timespan_maker,
        fuse_groups=True,
    ),
    guitar=ersilia.guitar_tremolo_musicSpecifier,
)

segment_maker.add_setting(
    timespan_identifier=[
        -2, 1,
        -1, 1,
        -3, 1,
        -1, 1,
        -1, 1,
        -3, 1,
        -1, 1,
        -1, 1,
    ],
    timespan_maker=new(
        ersilia.sparse_timespan_maker,
        fuse_groups=True,
    ),
)
saxophone = ersilia.wind_tremolo_music_specifier
    .transpose(12),
piano_rh = ersilia.piano_tremolo_music_specifier
    .transpose(24),
}

### AGITATO ###

segment_maker.add_setting{
    timespan_identifier=[
        -2, 1,
        -2, 1,
        -2, 2,
        -3, 1,
        -2, 3,
        -3, 1,
        -2,
        -2,
    ],
    timespan_maker=new(ersilia.dense_timespan_maker,
        padding=(1, 4),
    ),
    percussion=consort.MusicSpecifierSequence(
        application_rate='division',
        music_specifiers=[
            ersilia.percussion_tom_fanfare_music_specifier,
            ersilia.percussion_temple_block_fanfare_music_specifier,
            ersilia.percussion_tom_fanfare_music_specifier,
            ersilia.percussion_tom_fanfare_music_specifier,
            ersilia.percussion_temple_block_fanfare_music_specifier,
            ersilia.percussion_temple_block_fanfare_music_specifier,
        ]
    ),
}

segment_maker.add_setting(
    timespan_identifier=[
        -1, 1,
        -2, 2,
        -3, 1,
        -2, 2,
        -3, 4,
        -4, 1,
    ],
    timespan_maker=ersilia.dense_timespan_maker,
    piano_rh=ersilia.piano_agitato_music_specifier
        .rotate(1),
    piano_lh=ersilia.piano_agitato_music_specifier
        .rotate(2)
        .transpose(-24),
    saxophone=ersilia.saxophone_agitato_music_specifier
        .transpose('C2'),
}

967
segment_maker.add_setting(
    timespan_identifier=[
        -4, 1,
    ],
    timespan_maker=ersilia.sparse_timespan_maker
        .rotate(),
    flute=ersilia.wind_continuo_musicSpecifier,
    oboe=ersilia.wind_continuo_musicSpecifier,
    clarinet=ersilia.wind_continuo_musicSpecifier,
)

### POINTILLIST ###

segment_maker.add_setting(
    timespan_identifier=[
        -1, 4,
        -2, 1,
        -2, 2,
        -1,
    ],
    timespan_maker=ersilia.sparse_timespan_maker
        .rotate(),
    piano_rh=ersilia.piano_pointillist_musicSpecifier,
    piano_lh=ersilia.piano_pointillist_musicSpecifier
        .transpose(-12),
)

segment_maker.add_setting(
    timespan_identifier=[
        -1, 1,
        -3, 1,
        -2, 2,
        -1, 3,
        -2, 1,
        -1, 1,
        -2,
    ],
    timespan_maker=ersilia.dense_timespan_maker,
    flute=ersilia.wind_pointillist_musicSpecifier,
    oboe=ersilia.wind_pointillist_musicSpecifier
        .rotate(),
    clarinet=ersilia.wind_pointillist_musicSpecifier
        .transpose(-12)
        .rotate(2),
    piano_rh=ersilia.piano_palm_cluster_musicSpecifier,
)

segment_maker.add_setting(
    timespan_identifier=[
        -3, 1,
        -2, 1,
    ],
timespan_maker = new(
    ersilia.dense_timespan_maker,
    timespanSpecifier__minimum_duration=(1, 8),
),

violin=ersilia.string_pointillist_musicSpecifier,
viola=ersilia.string_pointillist_musicSpecifier
    .transpose('C3'),
  cello=ersilia.string_pointillist_musicSpecifier
    .transpose('C2'),
bass=ersilia.string_pointillist_musicSpecifier
    .transpose('E1'),
)

### AUXILIARY ###

musicSpecifier = consort.MusicSpecifierSequence(
    application_rate='division',
    musicSpecifiers=[
        ersilia.shaker_sporadic_musicSpecifier,
        ersilia.shaker_tremolo_musicSpecifier,
        ersilia.shaker_sporadic_musicSpecifier,
        ersilia.shaker_decelerando_musicSpecifier,
    ],
)
timespan_maker = new(
    ersilia.sparse_timespan_maker,
    padding=(3, 4),
)

segment_maker.add_setting(
    timespan_identifier=[
        -1, 1,
        -2, 2,
        -3, 1,
        -1, 1,
        -2, 1,
        -3, 1,
        -2,
    ],
)
timespan_maker=timespan_maker,
    flute=musicSpecifier,
    clarinet=musicSpecifier,
    oboe=musicSpecifier,
)

segment_maker.add_setting(
    timespan_identifier=[
        -2, 1,
        -3, 1,
        -1, 1,
    ],
timespan_maker=timespan_maker,
violin=music_specifier,
viola=music_specifier,
cello=music_specifier,
}

### CUT THROUGH ###

segment_maker.add_setting(
    timespan_identifier=[
        -8, 1,
        -13, 1
    ],
    timespan_maker=new(
        ersilia.dense_timespan_maker,
    ),
    flute=ersilia.wind_agitato_music_specifier
        .rotate(),
    clarinet=ersilia.wind_agitato_music_specifier.transpose('C2')
        .rotate(2),
    oboe=ersilia.wind_agitato_music_specifier
        .rotate(3),
    saxophone=ersilia.saxophone_agitato_music_specifier.transpose('C2'),
    guitar=ersilia.guitar_agitato_music_specifier,
    piano_rh=ersilia.piano_agitato_music_specifier
        .rotate(),
    piano_lh=ersilia.piano_agitato_music_specifier
        .transpose(-24)
        .rotate(2),
    violin=ersilia.string_agitato_music_specifier
        .rotate(),
    viola=ersilia.string_agitato_music_specifier
        .transpose('C3')
        .rotate(2),
    cello=ersilia.string_agitato_music_specifier
        .transpose('C2')
        .rotate(3),
    bass=ersilia.string_agitato_music_specifier
        .transpose('E1')
        .rotate(4),
    )

segment_maker.add_setting(
    timespan_identifier=[
        -4, 1,
        -5, 1,
        -6, 1,
        -2, 1,
    ],
timespan_maker = new(ersilia.sparse_timespan_maker,
padding=(1, 2),
repeat=False,
),
percussion = ersilia.percussion_crotale_flash_music_specifier,
)

### INTERRUPT ###

segment_maker.add_setting(
timespan_identifier=[
3, -1,
2, -1,
1,
],
timespan_maker=ersilia.tutti_timespan_maker,
piano_lh=ersilia.piano_arm_cluster_musicSpecifier
.transpose(-12),
percussion=ersilia.percussion_snare_interruption_musicSpecifier,
)

segment_maker.add_setting(
timespan_maker=consort.BoundaryTimespanMaker(
labels='piano arm cluster',
output_masks=[
    rhythmmakertools.SilenceMask(
        indices=[0, 1, 3],
        period=5,
    ),
],
start_talea=(3, 8),
),
guitar=new(ersilia.guitar_strummed_musicSpecifier,
attachment_handler__dynamic_expressions=consort.DynamicExpression(
    dynamic_tokens='f',
    only_first=True,
),
),
)

segment_maker.add_setting(
timespan_identifier=[-1, 1],
timespan_maker=consort.BoundaryTimespanMaker(
labels='piano arm cluster',
output_masks=[
    rhythmmakertools.SilenceMask(
        indices=[0, 1, 3],
        period=5,
    ),
],
)
D.3.5 ersilia.segments.sort

```python
# -*- encoding: utf-8 -*-
import abjad
import consort
import ersilia
from abjad import new
from abjad.tools import rhythmtools
from abjad.tools import timespantools

segment_maker = ersilia.ErsiliaSegmentMaker(
    desired_duration_in_seconds=150,
    name='Sort',
    permitted_time_signatures=ersilia.permitted_time_signatures,
    tempo=abjad.Tempo((1, 4), 64),
)

### PEDAL ###

segment_maker.add_setting(
    timespan_identifier=[
        1,
        -1, 1,
        -2, 1,
    ],
    timespan_maker=new(ersilia.sustained_timespan_maker,
        fuse_groups=True,
        initial_silence_talea=None,
    ),
    clarinet=ersilia.wind_low_pedal_musicSpecifier
        .transpose(12),
    saxophone=ersilia.wind_low_pedal_musicSpecifier
        .transpose(12),
    percussion=ersilia.percussion_low_pedal_musicSpecifier,
    bass=ersilia.string_low_pedal_musicSpecifier
        .transpose('E1'),
)

### TREMOLO ###
```
segment_maker.add_setting(
    timespan_maker=new(
        ersilia.sustained_timespan_maker,
        fuse_groups=True,
    ),
    guitar=ersilia.guitar_undulation_tremolo_musicSpecifier,
    piano_rh=ersilia.piano_tremolo_musicSpecifier,
)

segment_maker.add_setting(
    timespan_identifier=[-14, 3],
    timespan_maker=new(
        ersilia.sustained_timespan_maker,
        fuse_groups=True,
    ),
    percussion=ersilia.percussion_marimba_tremolo_musicSpecifier,
)

segment_maker.add_setting(
    timespan_identifier=[
        -1, 1,
        -1, 1,
        -3, 1,
        -1, 1,
        -2, 1,
        -1, 1,
        -3, 1,
    ],
    timespan_maker=new(
        ersilia.sparse_timespan_maker,
        fuse_groups=True,
    ),
    saxophone=ersilia.wind_tremolo_musicSpecifier
        .transpose(12),
    piano_rh=ersilia.piano_tremolo_musicSpecifier
        .transpose(24),
)

segment_maker.add_setting(
    timespan_maker=new(
        ersilia.sparse_timespan_maker,
        silence_talea__denominator=4,
    ),
    violin=ersilia.string_tremolo_musicSpecifier,
    viola=new(
        ersilia.string_tremolo_musicSpecifier,
        pitch_handler__register_specifier__base_pitch='C3',
    ),
    cello=new(
        ersilia.string_tremolo_musicSpecifier,
        pitch_handler__register_specifier__base_pitch='C2',
    ),
)
### AGITATO ###

```python
segment_maker.add_setting(
    timespan_identifier=[[
        -2, 1,
        -1, 1,
        -2, 3,
        -2, 2,
        -1, 3,
        -2, 1,
        -2,
        ],
    timespan_maker=ersilia.dense_timespan_maker,
    percussion=ersilia.percussion_marimba_agitato_musicSpecifier,
)

segment_maker.add_setting(
    timespan_identifier=[[
        -1, 1,
        -2, 2,
        -3, 1,
        -2, 2,
        -3, 4,
        -4, 1,
        ],
    timespan_maker=ersilia.dense_timespan_maker,
    percussion=ersilia.percussion_temple_block_fanfare_musicSpecifier,
    piano_rh=ersilia.piano_agitato_musicSpecifier
       .rotate(1),
    piano_lh=ersilia.piano_agitato_musicSpecifier
       .rotate(2)
       .transpose(-24),
    saxophone=ersilia.saxophone_agitato_musicSpecifier
       .transpose('C2'),
)

### CONTINUO ###

segment_maker.add_setting(
    timespan_identifier=[[
        -2, 3,
        -4, 2,
        -3, 1,
        ],
    timespan_maker=new(
        ersilia.sparse_timespan_maker,
        padding=(1, 2),
    ),
    percussion=new(
        ersilia.percussion_crotalesFlash_musicSpecifier,
        rhythm_maker__inciseSpecifier__prefix_counts=[1],
    ),
)
musicSpecifier = consort.MusicSpecifierSequence(
    application_rate='division',
    music_specifiers=[
        ersilia.wind_continuo_music_specifier,
        ersilia.wind_continuo_music_specifier,
        ersilia.saxophone_agitato_music_specifier.transpose(24),
        ersilia.wind_continuo_music_specifier,
        ersilia.saxophone_agitato_music_specifier.transpose(24),
        ersilia.saxophone_agitato_music_specifier.transpose(24),
    ],
)
segment_maker.add_setting(
    timespan_identifier=[
        -9, 1,
        -4, 1,
        -2, 3,
        -1, 1,
        -4, 2,
        -2, 3,
        -8, 1,
        -2, 3,
        -4, 2,
        -3, 1,
    ],
    timespan_maker=ersilia.dense_timespan_maker
        .rotate(),
            flute=musicSpecifier,
            oboe=musicSpecifier.transpose(12),
            clarinet=musicSpecifier,  # .transpose(-12),
)

### POINTILLIST ###

segment_maker.add_setting(
    timespan_identifier=[
        -1, 1,
        -3, 1,
        -2, 2,
        -1, 3,
        -2, 1,
        -1, 1,
        -2,
    ],
    timespan_maker=ersilia.sparse_timespan_maker,
            flute=ersilia.wind_pointillist_music_specifier,
            oboe=new(
                ersilia.wind_pointillist_music_specifier,
                pitch_handler__register_specifier__base_pitch='G4',
            ),
            clarinet=ersilia.wind_pointillist_music_specifier
                .transpose(-12)
                .rotate(2),
            saxophone=ersilia.wind_pointillist_music_specifier
                .transpose(-12)
rotate(2),
piano_rh=ersilia.piano_palm_cluster_musicSpecifier,
)

segment_maker.add_setting(
timespan_identifier=[
    -3, 1,
    -2, 1,
    -1, 4,
    -2, 1,
    -4, 1,
    -1, 1,
    -2, 1,
],
timespan_maker=new(
ersilia.sparse_timespan_maker,
timespanSpecifier._minimum_duration=(1, 8),
),

violin=ersilia.string_legato_musicSpecifier,
viola=ersilia.string_legato_musicSpecifier
    .transpose('C3'),
    cello=ersilia.string_legato_musicSpecifier
    .transpose('C2'),
bass=ersilia.string_legato_musicSpecifier
    .transpose('E1'),
)

### AUXILIARY ###

musicSpecifier = new(
ersilia.pitch_pipe_musicSpecifier,
rhythmMaker._output_masks=[rhythmMakerTools.SustainMask(indices=[0, -1])],
attachmentHandler._dynamic_expressions=consort.DynamicExpression(
    start_dynamic_tokens='fp',
    stop_dynamic_tokens='o',
),
)

segment_maker.add_setting(
timespan_identifier=timespantools.Timespan(0, (3, 2)),
timespan_maker=new(
ersilia.tutti_timespan_maker,
    fuse_groups=True,
timespanSpecifier=consort.TimespanSpecifier(
        minimum_duration=0,
    ),
guitar_pp=musicSpecifier,
    piano_pp=musicSpecifier,
    percussion_pp=musicSpecifier,
    bass_pp=musicSpecifier,
),

segment_maker.add_setting(
timespan_identifier=[

timespan_maker = new(ersilia.dense_timespan_maker,
  fuse_groups=True,
  silence_talea_denominator=4,
  timespanSpecifier=consort.TimespanSpecifier(  
    minimum_duration@0,  
  ),
  ),
  guitar_pp=musicSpecifier,
  piano_pp=musicSpecifier,
  percussion_pp=musicSpecifier,
  bass_pp=musicSpecifier,
)

### INTERRUPT ###

segment_maker.add_setting(
  timespan_identifier=timespantools.Timespan(0, (1, 4)),
  piano_lh=ersilia.piano_arm_cluster_musicSpecifier
  .transpose(-12),
)

segment_maker.add_setting(
  timespan_identifier=[3, -1, 2, -1, 1],
  timespan_maker=ersilia.sparse_timespan_maker,
  percussion=ersilia.percussion_bamboo_windchimes_musicSpecifier,
)

segment_maker.add_setting(
  timespan_identifier=[-1, 3, -1, 3],
  timespan_maker=ersilia.sparse_timespan_maker,
  guitar=ersilia.guitar_strummed_musicSpecifier,
)

segment_maker.add_setting(
  timespan_identifier=[-1, 1],
  timespan_maker=consort.BoundaryTimespanMaker(  
    labels='bamboo windchimes',  
    output_masks=[  
      rhythmmakertools.SilenceMask(  
        indices=[0, 1, 3],
      ),
    ],
  ),
  percussion=ersilia.percussion_bamboo_windchimes_musicSpecifier,
  guitar=ersilia.guitar_strummed_musicSpecifier,
)
period=5,
)

start_talea=rhythmmakertools.Talea(
counts=[2, 3, 4],
denominator=8,
)

start_groupings=[3, 4, 3, 2],

violin=ersilia.string_overpressure_musicSpecifier,

viola=ersilia.string_overpressure_musicSpecifier

.transpose(7)

.rotate(),

cello=ersilia.string_overpressure_musicSpecifier

.transpose(-7)

.rotate(),
)

segment_maker.addSetting(
timespan_identifier=[
    -5, 1,
    -4, 1,
    -13, 1,
    -5,
]

timespan_maker=new(
    ersilia.dense_timespan_maker,
    fuse_groups=True,
    repeat=False,
),

percussion=ersilia.percussion_crotales_interruption_musicSpecifier,

silenced_contexts=segment_maker.score_template.all_voice_names,
)

segment_maker.addSetting(
timespan_identifier=timespantools.Timespan((321, 8), (325, 8)),

percussion=ersilia.percussion_crotales_flash_musicSpecifier,

silenced_contexts=segment_maker.score_template.all_voice_names,
)

D.4 ersilia stylesheet source

D.4.1 stylesheet.ily

\include "scheme.ily"

#(set-default-paper-size "11x17" 'portrait)

#(set-global-staff-size 12)

\header {

    composer = \markup {

        \column {

            \override #'(font-name . "Didot")

            \fontsize #3 "Josiah Wolf Oberholtzer (1984)"

            \tie...

            \rest...

            \note d'..."
\column { 
\center-align { 
\override #'(font-name . "Didot Italic") 
\fontsize #4 { 
  \line { Invisible Cities (iii): } 
} 
\vspace #0.5 
\override #'(font-name . "Didot") 
\fontsize #16 { 
  \line { ERSILIA } 
} 
\vspace #0.5 
\override #'(font-name . "Didot Italic") 
\fontsize #2 { 
  \line { ( a botanical survey of the uninhabited northeastern isles ) } 
} 
\vspace #1 
\override #'(font-name . "Didot Italic") 
\fontsize #4 { 
  \line { for Ensemble Dal Niente } 
  \null 
  \null 
} 
} 
} 
\paper { 
indent = 20\mm 
short-indent = 15\mm 
bottom-margin = 10\mm 
left-margin = 10\mm 
right-margin = 10\mm 
top-margin = 10\mm 
oddHeaderMarkup = \markup {} 
evenHeaderMarkup = \markup {} 
oddFooterMarkup = \markup {} 
\fill-line { 
\override #'(font-name . "Didot") 
\bold \fontsize #3 "Invisible Cities (iii): Ersilia" 
\override #'(font-name . "Didot") 
\bold \fontsize #3 \date 
\concat { 
  \override #'(font-name . "Didot") 
  \bold \fontsize #3
\on-the-fly \print-page-number-check-first
\fromproperty #'page:page-number-string

evenFooterMarkup = \markup
\fill-line {
  \concat {
  \override #'(font-name . "Didot")
  \on-the-fly \print-page-number-check-first
  \fromproperty #'page:page-number-string
  }
  \override #'(font-name . "Didot")
  \bold \fontsize #3 \date
  \override #'(font-name . "Didot")
  \bold \fontsize #3 "Invisible Cities (iii): Ersilia"
}
print-first-page-number = ##t
print-page-number = ##t
max-systems-per-page = 1
page-breaking = #ly:optimal-breaking
ragged-bottom = ##f
ragged-last-bottom = ##t
markup-system-spacing = #'(
  (basic-distance . 0)
  (minimum-distance . 12)
  (padding . 0)
  (stretchability . 0)
)

system-system-spacing = #'(
  (basic-distance . 12)
  (minimum-distance . 18)
  (padding . 12)
  (stretchability . 20)
)

top-markup-spacing = #'(
  (basic-distance . 0)
  (minimum-distance . 0)
  (padding . 0)
  (stretchability . 0)
)

top-system-spacing = #'(
  (basic-distance . 0)
  (minimum-distance . 10)
  (padding . 0)
  (stretchability . 0)
)

\layout {
  \accidentalStyle modern-cautionary
  ragged-bottom = ##f
  ragged-last = ##f
  ragged-right = ##t
}
%%% ANNOTATIONS %%%

\context {
  \Voice
  \name AnnotatedDivisionsVoice
  \type Engraver_group
  \alias Voice
  \override Accidental.stencil = ##f
  \override Dots.stencil = ##f
  \override Flag.stencil = ##f
  \override NoteCollision.merge-differently-dotted = ##t
  \override NoteCollision.merge-differently-headed = ##t
  \override NoteColumn.ignore-collision = ##t
  \override NoteHead.no-ledgers = ##t
  \override NoteHead.transparent = ##t
  \override Stem.stencil = ##f
  \override TupletBracket.direction = #down
  \override TupletBracket.outside-staff-padding = 1
  \override TupletBracket.outside-staff-priority = 999
  \override TupletBracket.thickness = 2
  \override TupletNumber.stencil = ##f
}

\context {
  \Voice
  \name AnnotatedPhrasesVoice
  \type Engraver_group
  \alias Voice
  \override Accidental.stencil = ##f
  \override Dots.stencil = ##f
  \override Flag.stencil = ##f
  \override NoteCollision.merge-differently-dotted = ##t
  \override NoteCollision.merge-differently-headed = ##t
  \override NoteColumn.ignore-collision = ##t
  \override NoteHead.no-ledgers = ##t
  \override NoteHead.transparent = ##t
  \override Stem.stencil = ##f
  \override TupletBracket.direction = #down
  \override TupletBracket.outside-staff-padding = 1
  \override TupletBracket.outside-staff-priority = 1000
  \override TupletBracket.thickness = 2
  \override TupletNumber.stencil = ##f
}

%%% DEFAULTS %%%

\context {
  \Voice
  \remove Forbid_line_break_engraver
}
\context {
  \Staff

981
\remove Time_signature_engraver
\accepts AnnotatedDivisionsVoice
\accepts AnnotatedPhrasesVoice
}

\context {
\Dynamics
\remove Bar_engraver
\override DynamicLineSpanner.staff-padding = 11.5
\override DynamicText.self-alignment-X = -1
}

%%% TIME SIGNATURE CONTEXT %%%

\context {
\name TimeSignatureContext
\type Engraver_group
\consists Axis_group_engraver
\consists Bar_number_engraver
\consists Mark_engraver
\consists Metronome_mark_engraver
\consists Script_engraver
\consists Text_engraver
\consists Text_spanner_engraver
\consists Time_signature_engraver
\override BarNumber.extra-offset = #'(-6 . -4)
\override BarNumber.font-name = "Didot Italic"
\override BarNumber.font-size = 1
\override BarNumber.padding = 4
\override MetronomeMark.X-extent = #'(0 . 0)
\override MetronomeMark.Y-extent = #'(0 . 0)
\override MetronomeMark.break-align-symbols = #'(left-edge)
\override MetronomeMark.extra-offset = #'(0 . 2)
\override MetronomeMark.font-size = 3
\override MetronomeMark.use-skylines = ##f
\override MetronomeMark.padding = 0
\override MetronomeMark.staff-padding = 0
\override MetronomeMark.outside-staff-padding = 0
\override MetronomeMark.outside-staff-horizontal-padding = 0
\override MetronomeMark.minimum-space = 0
\override RehearsalMark.X-extent = #'(0 . 0)
\override RehearsalMark.X-offset = 6
\override RehearsalMark.Y-offset = -2.25
\override RehearsalMark.break-align-symbols = #'(time-signature)
\override RehearsalMark.break-visibility = #end-of-line-invisible
\override RehearsalMark.font-name = "Didot"
\override RehearsalMark.font-size = 10
\override RehearsalMark.outside-staff-priority = 500
\override RehearsalMark.self-alignment-X = #center
\override Script.extra-offset = #'(4 . -9)
\override Script.font-size = 6
\override TextScript.font-size = 3
\override TextScript.outside-staff-priority = 600
\override TextScript.padding = 6
\override TextScript.parent-alignment-X = #center
\override TextScript.self-alignment-X = #center
\override TextSpanner.bound-details.right.attach-dir = #LEFT
\override TextSpanner.padding = 6.75
\override TimeSignature.X-extent = #'(0 . 0)
\override TimeSignature.break-align-symbol = #'left-edge
\override TimeSignature.break-visibility = #end-of-line-invisible
\override TimeSignature.font-size = 3
\override TimeSignature.space-alist.clef = #'(extra-space . 0.5)
\override TimeSignature.style = #'numbered
\override VerticalAxisGroup.default-staff-staff-spacing = #'(
  (basic-distance . 0)
  (minimum-distance . 15)
  (padding . 8)
  (stretchability . 0)
)
\override VerticalAxisGroup.minimum-Y-extent = #'(-20 . 20)
#endif

%% PERFORMERS %%

\context {
  \Staff
  \name PitchPipes
  \type Engraver_group
  \alias Staff
  \RemoveEmptyStaves
  \override StaffSymbol.line-count = 1
}

\context {
  \Staff
  \name FluteStaff
  \type Engraver_group
  \alias Staff
}

\context {
  \Staff
  \name ClarinetStaff
  \type Engraver_group
  \alias Staff
}

\context {
  \Staff
  \name OboeStaff
  \type Engraver_group
  \alias Staff
}
\context { 
  \Staff  
  \name SaxophoneStaff  
  \type Engraver_group  
  \alias Staff  
}  

\context {  
  \StaffGroup  
  \name WindSectionStaffGroup  
  \type Engraver_group  
  \alias StaffGroup  
  \accepts FluteStaff  
  \accepts ClarinetStaff  
  \accepts OboeStaff  
  \accepts SaxophoneStaff  
  \override StaffGrouper.staffgroup-staff-spacing = #'(( 
    (basic-distance . 0)  
    (minimum-distance . 20)  
    (padding . 15)  
    (stretchability . 10)  
  )  
)  

\context {  
  \Staff  
  \name GuitarStaff  
  \type Engraver_group  
  \alias Staff  
}  

\context {  
  \StaffGroup  
  \name GuitarStaffGroup  
  \type Engraver_group  
  \alias StaffGroup  
  \accepts GuitarStaff  
  \accepts PitchPipes  
  systemStartDelimiter = '#SystemStartSquare  
}  

\context {  
  \Staff  
  \name PianoUpperStaff  
  \type Engraver_group  
  \alias Staff  
}  

\context {  
  \Staff  
  \name PianoLowerStaff  
  \type Engraver_group  
  \alias Staff  
}
\context{
\PianoStaff
\remove "Keep_alive_together_engraver"
\accepts PianoLowerStaff
\accepts PianoUpperStaff
\accepts PitchPipes
}

\context{
\StaffGroup
\name PianoStaffGroup
\type Engraver_group
\alias StaffGroup
\accepts PianoStaff
\accepts PitchPipes
\systemStartDelimiter = #'SystemStartSquare
}

\context{
\Staff
\name PercussionStaff
\type Engraver_group
\alias Staff
}

\context{
\StaffGroup
\name PercussionStaffGroup
\type Engraver_group
\alias StaffGroup
\accepts PercussionStaff
\accepts PitchPipes
\override StaffGrouper.staffgroup-staff-spacing = #'(
  (basic-distance . 0)
  (minimum-distance . 20)
  (padding . 15)
  (stretchability . 10)
)
\systemStartDelimiter = #'SystemStartSquare
}

\context{
\StaffGroup
\name PercussionSectionStaffGroup
\type Engraver_group
\alias StaffGroup
\accepts GuitarStaffGroup
\accepts PianoStaffGroup
\accepts PercussionStaffGroup
}

\context {
\Staff
\name ViolinStaff
\type Engraver_group
\alias Staff

\context {
\Staff
\name ViolaStaff
\type Engraver_group
\alias Staff
}

\context {
\Staff
\name CelloStaff
\type Engraver_group
\alias Staff
}

\context {
\Staff
\name ContrabassStaff
\type Engraver_group
\alias Staff
}

\context {
\StaffGroup
\name ContrabassStaffGroup
\type Engraver_group
\alias StaffGroup
\accepts ContrabassStaff
\accepts PitchPipes
systemStartDelimiter = #'SystemStartSquare
}

\context {
\StaffGroup
\name StringSectionStaffGroup
\type Engraver_group
\alias StaffGroup
\accepts ViolinStaff
\accepts ViolaStaff
\accepts CelloStaff
\accepts ContrabassStaffGroup
}

%% SCORE %%

\context {
\Score
\accepts PercussionSectionStaffGroup
\accepts StringSectionStaffGroup

\accepts TimeSignatureContext
\accepts WindSectionStaffGroup
\remove Bar_number_engraver
\remove Mark_engraver
\remove Metronome_mark_engraver
\override BarLine.bar-extent = #'(-2 . 2)
\override BarLine.hair-thickness = 0.5
\override BarLine.space-alist = #'(
  (time-signature extra-space . 0.0)
  (custos minimum-space . 0.0)
  (clef minimum-space . 0.0)
  (key-signature extra-space . 0.0)
  (key-cancellation extra-space . 0.0)
  (first-note fixed-space . 0.0)
  (next-note semi-fixed-space . 0.0)
  (right-edge extra-space . 0.0)
)
\override Beam.beam-thickness = 0.75
\%\override Beam.direction = #down
\override Beam.breakable = ##t
\override Beam.damping = 5
\override Beam.length-fraction = 1.5
\override Glissando.breakable = ##t
\override Glissando.thickness = 3
\override Hairpin.bound-padding = 1.5
\override InstrumentName.self-alignment-X = #RIGHT
\override MultiMeasureRest.expand-limit = #1
\override NoteCollision.merge-differently-dotted = ##t
\override NoteColumn.ignore-collision = ##t
\override OttavaBracket.outside-staff-priority = 500
\override OttavaBracket.padding = 2
\shape #'((-1.5 . 0) (-1 . 0) (-0.5 . 0) (0 . 0)) RepeatTie
\override RepeatTie.X-extent = ##f
\override SpanBar.hair-thickness = 0.5
\override SpacingSpanner.base-shortest-duration = #(ly:make-moment 1 32)
\override SpacingSpanner.strict-grace-spacing = ##f
\override SpacingSpanner.strict-note-spacing = ##f
\override StaffSymbol.color = #(x11-color 'grey50)
\override StaffSymbol.layer = -1
\override Stem.details.beamed-lengths = #'(6)
\override Stem.details.lengths = #'(6)
\override Stem.stemlet-length = 1.5
\override StemTremolo.beam-width = 1.5
\override StemTremolo.flag-count = 4
\override StemTremolo.slope = 0.5
\override SustainPedal.self-alignment-X = #CENTER
\override SustainPedallineSpanner.padding = 2
\override SustainPedallineSpanner.outside-staff-padding = 2
\override SustainPedallineSpanner.to-barline = ##t
\override SystemStartSquare.thickness = 2
\override TextSpanner.padding = 1
\override TextSpanner.bound-details.right.padding = 2
\override TrillSpanner.bound-details.right.padding = 1
\override TupletBracket.breakable = \##t
\override TupletBracket.full-length-padding = 1.5
\override TupletBracket.full-length-to-extent = \##f
\override TupletBracket.padding = 1.5
\override TupletBracket.outside-staff-padding = 0.75
\override TupletNumber.font-size = 1
\override TupletNumber.text = \tuplet-number::calc-fraction-text
\override StaffGrouper.staffgroup-staff-spacing = #'( (basic-distance . 10) (minimum-distance . 10) (padding . 5) (stretchability . 0) )
\override StaffGrouper.staff-staff-spacing = #'( (basic-distance . 10) (minimum-distance . 10) (padding . 5) (stretchability . 0) )
autoBeaming = \##f
pedalSustainStyle = #'mixed
proportionalNotationDuration = #(ly:make-moment 1 32)
tupletFullLength = \##t
barNumberFormatter = #format-oval-barnumbers
)
}
References


This thesis was typeset using \LaTeX, originally developed by Leslie Lamport and based on Donald Knuth’s \TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont’s humanist typeface. The monospaced text is set in 10 point Inconsolata. The interactive Python interpreter sessions and their graphic output were managed by \textit{abjad-book}, a tool for executing and embedding Python code and music notation in \LaTeX documents. Notation examples were rendered by LilyPond and graph-theoretic examples were rendered by Graphviz.