Abjad Documentation

Release 2.11

Trevor Bača, Josiah Oberholtzer, Víctor Adán

February 10, 2013

CONTENTS

Part I

Start here

CHAPTER

ONE

ABJAD?

Abjad is an interactive software system designed to help composers build up complex pieces of music notation in an iterative and incremental way. Use Abjad to create a symbolic representation of all the notes, rests, staves, tuplets, beams and slurs in any score. Because Abjad extends the Python programming language, you can use Abjad to make systematic changes to your music as you work. And because Abjad wraps the powerful LilyPond music notation package, you can use Abjad to control the typographic details of the symbols on the page.

1.1 Abjad extends LilyPond

LilyPond is an open-source music notation package invented by Han-Wen Nienhuys and Jan Niewenhuizen and extended by an international team of developers and musicians. LilyPond differs from other music engraving programs in a number of ways. LilyPond separates musical content from page layout. LiyPond affords typographic control over almost everything. And LilyPond implements a powerfully correct model of the musical score.

You can start working with Abjad right away because Abjad creates LilyPond files for you automatically. But you will work with Abjad faster and more effectively if you understand the structure of the LilyPond files Abjad creates. For this reason we recommend new users spend a couple of days learning LilyPond first.

Start by reading about text input in LilyPond. Then work the LilyPond tutorial. You can test your understanding of LilyPond by using the program to engrave of a Bach chorale. Use a grand staff and and include slurs, fermatas and so on. Once you can engrave a chorale in LilyPond you'll understand the way Abjad works with LilyPond behind the scenes.

1.2 Abjad extends Python

Python is an open-source programming language invented by Guido van Rossum and further developed by a team of programmers working in many countries around the world. Python is used to provision servers, process text, develop distributed systems and do much more besides. The dynamic language and interpreter features of Python are similar to Ruby while the syntax of Python resembles C, C++ and Java.

To get the most out of Abjad you need to know (or learn) the basics of programming in Python. Abjad extends Python because it makes no sense to reinvent the wheel modern programming langauges have developed to find, sort, store, model and encapsulate information. Abjad simply piggy-backs on the ways of doing these things that Python provides. So to use Abjad effectively you need to know the way these things are done in Python.

Start with the Python tutorial. The tutorial is structured in 15 chapters and you should work through the first 12. This will take a day or two and you'll be able to use all the information you read in the Python tutorial in Abjad. If you're an experienced programmer you should skip chapters 1 - 3 but read 4 - 12. When you're done you can give yourself the equivlanent of the chorale test suggested above. First open a file and define a couple of classes and functions in it. Then open a second file and write some code to first import and then do stuff with the classes and functions you defined in the first file. Once you can easily do this without looking at the Python docs you'll be in a much better position to work with Abjad.

1.3 What next?

The most important parts of Abjad are the interlocking objects that structure the system. Read about the way Abjad models pitch, duration, leaves, containers, spanners and marks in the *Abjad reference manual*.

But note that important parts of the system are missing from the manual. The reason for this is that we completed the Abjad API months before we started the manual. This means that classes and functions you look up in the API may not yet be documented in the manual. The reference manual will eventually document all parts of the system. But until then check the API if the manual doesn't yet have what you need.

Once you understand the basics about how to work with Abjad you should spend some time with the *Abjad API*. The API documents all the functionality available in the system. Abjad comprises about 168,000 lines of code. About half of these implement the automated tests that check the correctness of Abjad. The rest of the code implements 39 packages comprising 221 classes and 1029 functions. All of these are documented in the API.

1.4 Mailing lists

As you begin working with Abjad please be in touch.

Questions, comments and contributions are welcomed from composers everywhere.

Questions or comments? Join the abjad-user list.

Want to contribute? Join the abjad-devel list.

CHAPTER

TWO

INSTALLATION

2.1 Abjad depends on Python

You must have Python 2.7 installed to run Abjad.

Abjad does not yet support the Python 3.x series of releases.

To check the version of Python installed on your computer type the following:

```
python --version
```

You can download different versions of Python at http://www.python.org.

2.2 Abjad depends on LilyPond

You must have LilyPond 2.16 or greater installed for Abjad to work properly.

You can download LilyPond at http://www.lilypond.org.

After you have installed LilyPond you should type the following to see if LilyPond is callable from your commandline:

```
lilypond --version
```

If LilyPond is not callable from your commandline you should add the location of the LilyPond executable to your `PATH` environment variable.

If you are new to working with the commandline you should use Google to get a basic introduction to editing your profile and setting environment variables.

2.3 Installing the current packaged version of Abjad with `easy_install`

There are different ways to install Python packages on your computer. One of the most direct ways is with `easy_install`.

If you have `easy_install` installed on your computer then you can install Abjad with this command:

```
sudo easy_install -U abjad
```

Python will install Abjad in the site packages directory on your computer and you'll be ready to start using the system.

If you do not have `easy_install` installed on your computer then you should follow the instructions below to install the current packaged version of Abjad from the Python Package Index.

2.4 Installing the current packaged version of Abjad from the Python Package Index

If you do not have `easy_install` installed on your computer you should follow these steps to install the current packaged version of Abjad from the Python Package Index:

1. Download the current release of Abjad from http://pypi.python.org/pypi/Abjad.
2. Unarchive the downloaded file. Under MacOS and Windows you can double click the archived file.

   Under Linux execute the following command with `x.y` replaced by the current release of Abjad:

```
tar xzvf Abjad-x.y.tar.gz
```

3. Change into the directory created in step 2:

```
cd Abjad-x.y
```

4. Run the following under MacOS or Linux:

```
sudo python setup.py install
```

5. Or run this command under Windows after starting up a command shell with administrator privileges:

```
setup.py install
```

These commands will cause Python to install Abjad in your site packages directory. You'll then be ready to start using Abjad.

2.5 After install

When first run, Abjad creates an `.abjad` directory in your own `$HOME` directory. In `$HOME/.abjad` you will find the Abjad configuration file: `config.py`. Here you can tell Abjad about your preferred PDF file viewer, MIDI player, your preferred LilyPond language, etc. All relevant variables have defaults that you can change to suit your needs. In Linux, for example, you might want to set your `pdfviewer` to `evince` and your `MIDIplayer` to `tiMIDIty`.

`config.py` is a regular Python file, so you should make sure the file follows Python syntax.

2.6 Note for Linux users

Abjad defaults to `xdg-open` to display PDF files using your default PDF viewer. Most Linux distributions now come with `xdg-utils` installed.

If you do not have `xdg-utils` installed, you can download it from http://www.portland.freedsektop.org.

Alternatively you can set the `pdfviewer` variable in `$HOME/.abjad/config` to your favorite PDF viewer.

CHAPTER

THREE

VERSION HISTORY

3.1 Abjad 2.11

Released 2013-02-05. Built from r9468. Implements 515 public classes and 1016 functions totalling 210,000 lines of code.

3.1.1 The `MetricalHierarchy` class

A new `MetricalHierarchy` class is now available in the `timesignaturetools` package.

The class implements a rhythm tree-based model of nested time signature groupings.

The structure of the tree corresponds to factors of the time signature's numerator.

Each deeper level of the tree divides the previous by the next factor in sequence.

Prime divisions greater than `3` are converted to sequences of `2` and `3` summing to that prime. Hence `5` becomes `3+2` and `7` becomes `3+2+2`.

The `MetricalHierarchy` class models a common-practice understanding of meter:

```
>>> metrical_hierarchy = timesignaturetools.MetricalHierarchy((4, 4))
```

```
>>> metrical_hierarchy
MetricalHierarchy('(4/4 (1/4 1/4 1/4 1/4))')
```

```
>>> print metrical_hierarchy.pretty_rtm_format
(4/4 (
    1/4
    1/4
    1/4
    1/4))
```

```
>>> print timesignaturetools.MetricalHierarchy((3, 4)).pretty_rtm_format
(3/4 (
    1/4
    1/4
    1/4))
```

```
>>> print timesignaturetools.MetricalHierarchy((6, 8)).pretty_rtm_format
(6/8 (
    (3/8 (
        1/8
        1/8
        1/8))
    (3/8 (
        1/8
        1/8
        1/8))))
```

```
>>> print timesignaturetools.MetricalHierarchy((5, 4)).pretty_rtm_format
(5/4 (
    (3/4 (
        1/4
        1/4
        1/4))
    (2/4 (
        1/4
        1/4))))
```

```
>>> print timesignaturetools.MetricalHierarchy((5, 4),
...     decrease_durations_monotonically=False).pretty_rtm_format
(5/4 (
    (2/4 (
        1/4
        1/4))
    (3/4 (
        1/4
        1/4
        1/4))))
```

```
>>> print timesignaturetools.MetricalHierarchy((12, 8)).pretty_rtm_format
(12/8 (
    (3/8 (
        1/8
        1/8
        1/8))
    (3/8 (
        1/8
        1/8
        1/8))
    (3/8 (
        1/8
        1/8
        1/8))
    (3/8 (
        1/8
        1/8
        1/8))))
```

3.1.2 Rewriting rhythms according to a different metric hierarchy

A new `establish_metrical_hierarchy()` function is now available in the `timesignaturetools` package.

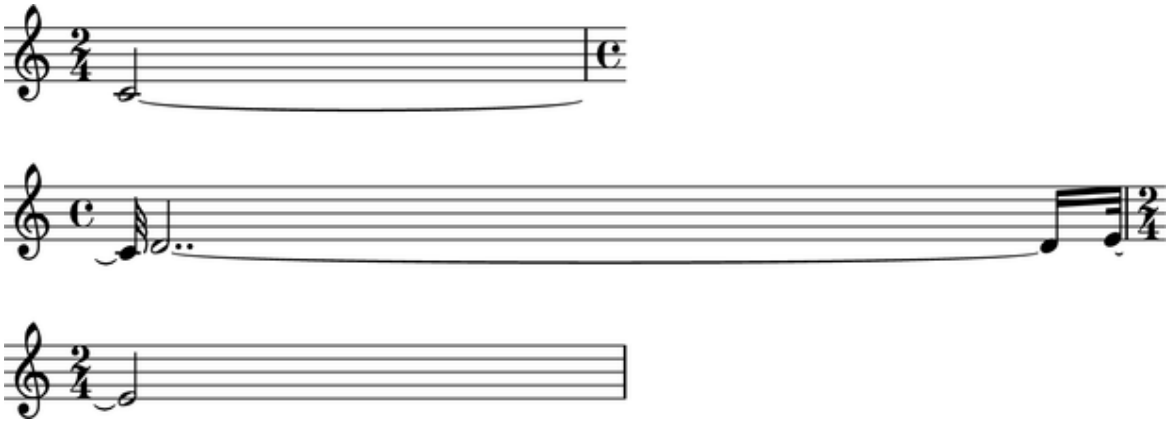
The function rewrites the contents of tie chains to match a metrical hierarchy.

Example 1. Rewrite the contents of a measure in a staff using the default metrical hierarchy for that measure's time signature:

```
>>> parseable = "abj: | 2/4 c'2 ~ || 4/4 c'32 d'2.. ~ d'16 e'32 ~ || 2/4 e'2 |"
```

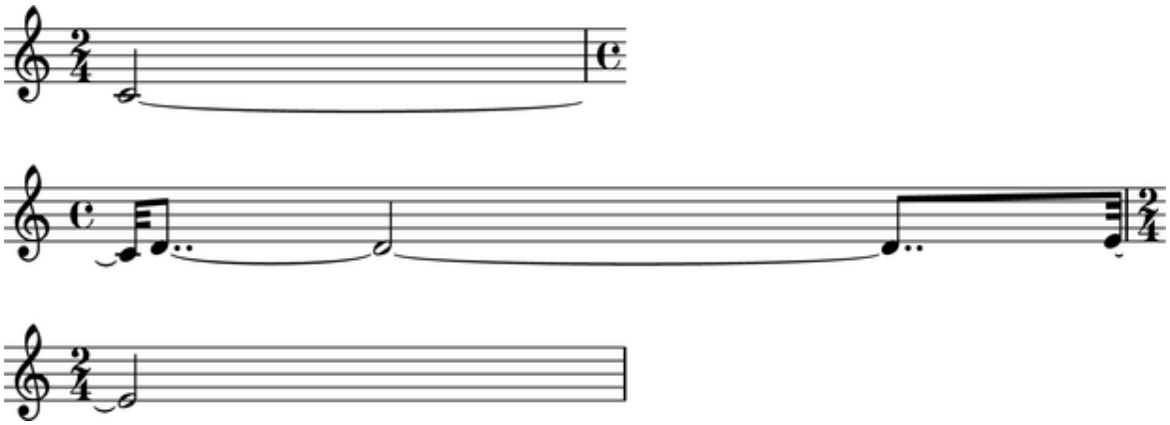
```
>>> staff = Staff(parseable)
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'2 ~
    }
    {
        \time 4/4
        c'32
        d'2.. ~
        d'16
        e'32 ~
    }
    {
        \time 2/4
        e'2
```

```
    }
}
```



```
>>> hierarchy = timesignaturetools.MetricalHierarchy((4, 4))
>>> print hierarchy.pretty_rtm_format
(4/4 (
    1/4
    1/4
    1/4
    1/4))
```

```
>>> timesignaturetools.establish_metrical_hierarchy(staff[1][:], hierarchy)
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'2 ~
    }
    {
        \time 4/4
        c'32
        d'8.. ~
        d'2 ~
        d'8..
        e'32 ~
    }
    {
        \time 2/4
        e'2
    }
}
```



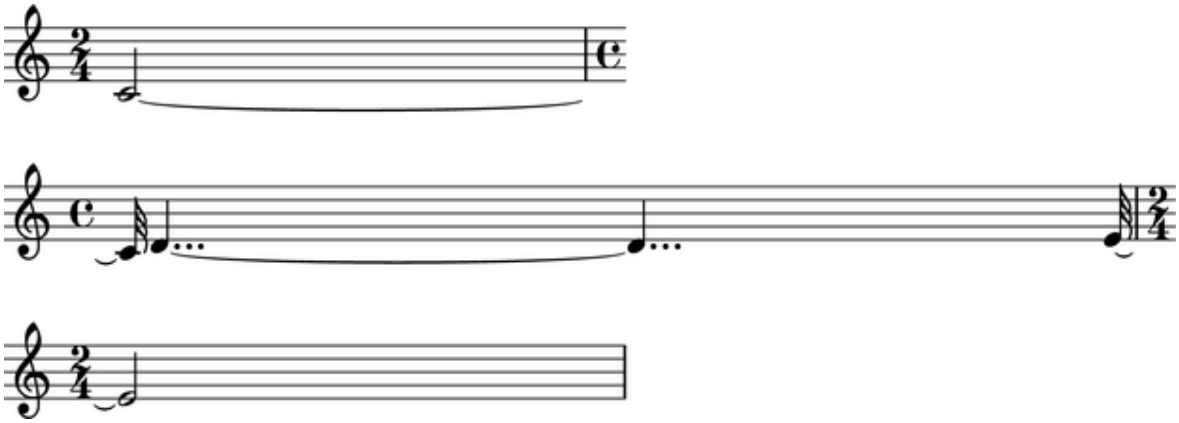
Example 2. Rewrite the contents of a measure in a staff using a custom metrical hierarchy:

```
>>> staff = Staff(parseable)
>>> f(staff)
\new Staff {
    {
```

```
        \time 2/4
        c'2 ~
    }
    {
        \time 4/4
        c'32
        d'2.. ~
        d'16
        e'32 ~
    }
    {
        \time 2/4
        e'2
    }
}
```

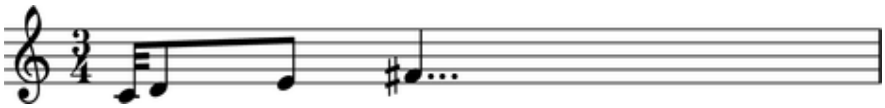
```
>>> rtm = '(4/4 ((2/4 (1/4 1/4)) (2/4 (1/4 1/4))))'
>>> hierarchy = timesignaturetools.MetricalHierarchy(rtm)
>>> print hierarchy.pretty_rtm_format
(4/4 (
    (2/4 (
        1/4
        1/4))
    (2/4 (
        1/4
        1/4))))
```

```
>>> timesignaturetools.establish_metrical_hierarchy(staff[1][:], hierarchy)
>>> f(staff)
\new Staff {
    {
        \time 2/4
        c'2 ~
    }
    {
        \time 4/4
        c'32
        d'4... ~
        d'4...
        e'32 ~
    }
    {
        \time 2/4
        e'2
    }
}
```



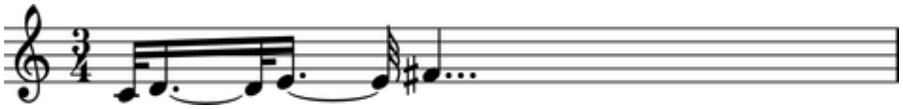
Example 3. Limit the maximum number of dots per leaf using *maximum_dot_count*:

```
>>> parseable = "abj: | 3/4 c'32 d'8 e'8 fs'4... |"
>>> measure = p(parseable)
>>> f(measure)
{
    \time 3/4
    c'32
    d'8
    e'8
    fs'4...
}
```



Do not constrain the *maximum_dot_count*:

```
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure)
>>> f(measure)
{
    \time 3/4
    c'32
    d'16. ~
    d'32
    e'16. ~
    e'32
    fs'4...
}
```

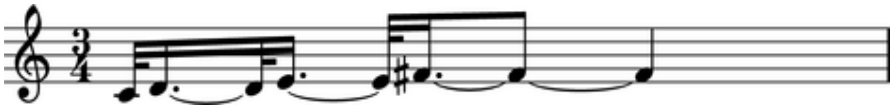


Constrain the *maximum_dot_count* to *2*:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure,
...     maximum_dot_count=2)
>>> f(measure)
{
    \time 3/4
    c'32
    d'16. ~
    d'32
    e'16. ~
    e'32
    fs'8.. ~
    fs'4
}
```

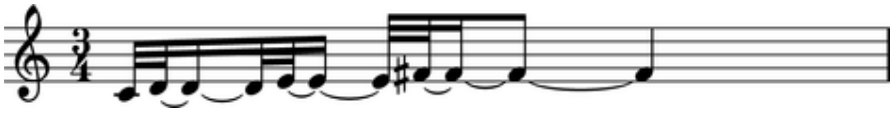
Constrain the *maximum_dot_count* to *1*:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure,
...     maximum_dot_count=1)
>>> f(measure)
{
    \time 3/4
    c'32
    d'16. ~
    d'32
    e'16. ~
    e'32
    fs'16. ~
    fs'8 ~
    fs'4
}
```



Constrain the *maximum_dot_count* to *0*:

```
>>> measure = p(parseable)
>>> timesignaturetools.establish_metrical_hierarchy(measure[:], measure,
...     maximum_dot_count=0)
>>> f(measure)
{
    \time 3/4
    c'32
    d'32 ~
    d'16 ~
    d'32
    e'32 ~
    e'16 ~
    e'32
    fs'32 ~
    fs'16 ~
    fs'8 ~
    fs'4
}
```



Many further examples are available in the API entry for the class.

3.1.3 The `quantizationtools` package

The `quantizationtools` package has been completely rewritten.

`Quantizer` quantizes sequences of attack-points in score trees.

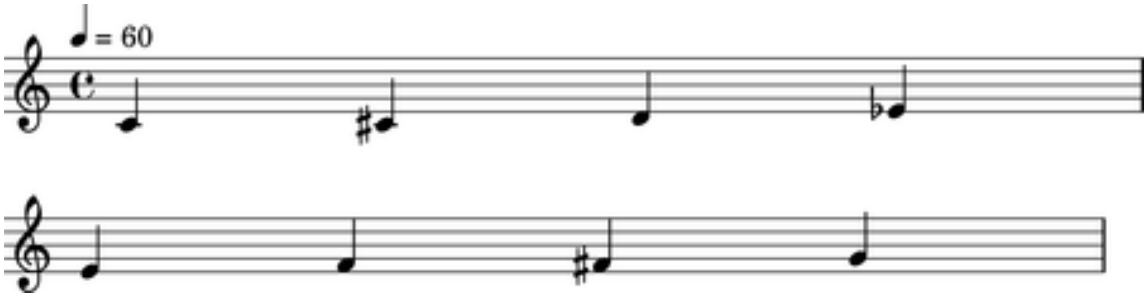
`QEventSequences` bundle attack-points together:

```
>>> quantizer = quantizationtools.Quantizer()
```

```
>>> durations = [1000] * 8
>>> pitches = range(8)
>>> q_event_sequence = quantizationtools.QEventSequence.from_millisecond_pitch_pairs(
...     zip(durations, pitches))
```

Quantization defaults to `4/4` output at `quarter=60`:

```
>>> result = quantizer(q_event_sequence)
>>> score = Score([Staff([result])])
>>> f(score)
\new Score <<
    \new Staff {
        \new Voice {
            {
                \time 4/4
                \tempo 4=60
                c'4
                cs'4
                d'4
                ef'4
            }
            {
                e'4
                f'4
                fs'4
                g'4
            }
        }
    }
>>
```



The behavior of the `Quantizer` can be modified at call-time.

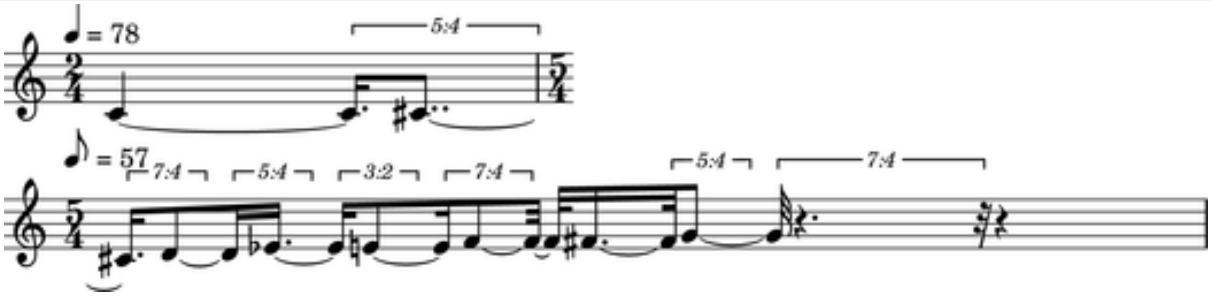
Pass a `QSchema` instance to alter the macro-structure of quantizer output.

Here, we quantize using settings specified by a `MeasurewiseQSchema`. This causes the `Quantizer` to group the output into measures with different tempi and time signatures:

```
>>> measurewise_q_schema = quantizationtools.MeasurewiseQSchema(
...     {'tempo': ((1, 4), 78), 'time_signature': (2, 4)},
...     {'tempo': ((1, 8), 57), 'time_signature': (5, 4)},
...     )
```

```
>>> result = quantizer(q_event_sequence, q_schema=measurewise_q_schema)
>>> score = Score([Staff([result])])
>>> f(score)
\new Score <<
    \new Staff {
        \new Voice {
            {
                \time 2/4
                \tempo 4=78
                c'4 ~
                \times 4/5 {
                    c'16.
                    cs'8.. ~
                }
            }
            {
                \time 5/4
                \tempo 8=57
                \times 4/7 {
                    cs'16.
                    d'8 ~
                }
                \times 4/5 {
```

```
                    d'16
                    ef'16. ~
                }
                \times 2/3 {
                    ef'16
                    e'8 ~
                }
                \times 4/7 {
                    e'16
                    f'8 ~
                    f'32 ~
                }
                f'32
                fs'16. ~
                \times 4/5 {
                    fs'32
                    g'8 ~
                }
                \times 4/7 {
                    g'32
                    r4. ~
                    r32 ~
                }
                r4
            }
        }
    }
>>
```

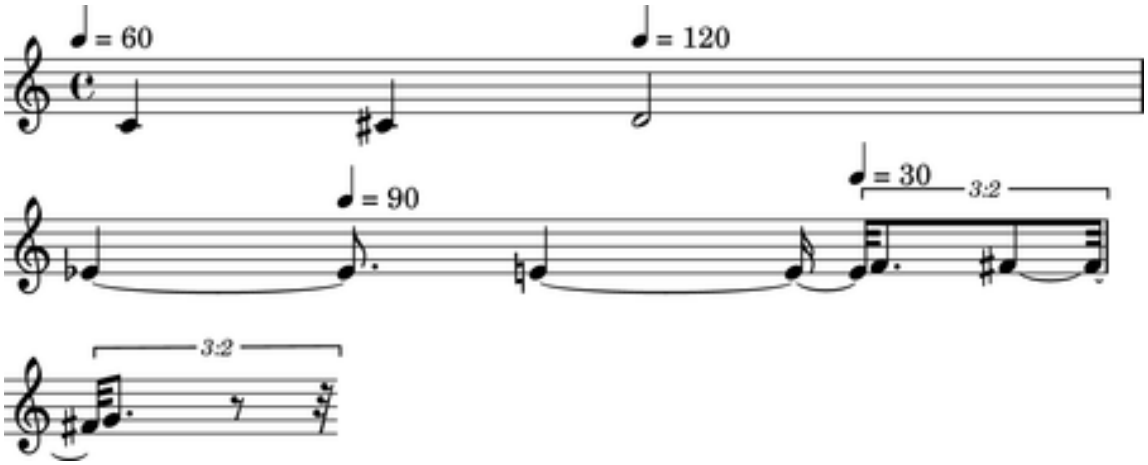


Here we quantize using settings specified by a `BeatwiseQSchema`. This keeps the output of the quantizer flattened and produces neither measures nor explicit time signatures. The default beatwise setting of `quarter=60` persists until the third beatspan:

```
>>> beatwise_q_schema = quantizationtools.BeatwiseQSchema(
... {
...     2: {'tempo': ((1, 4), 120)},
...     5: {'tempo': ((1, 4), 90)},
...     7: {'tempo': ((1, 4), 30)},
... })
```

```
>>> result = quantizer(q_event_sequence, q_schema=beatwise_q_schema)
>>> score = Score([Staff([result])])
>>> f(score)
\new Score <<
    \new Staff {
        \new Voice {
            \tempo 4=60
            c'4
            cs'4
            \tempo 4=120
            d'2
            ef'4 ~
            \tempo 4=90
            ef'8.
            e'4 ~
            e'16 ~
            \times 2/3 {
                \tempo 4=30
                e'32
                f'8.
```

```
                fs'8 ~
                fs'32 ~
            }
            \times 2/3 {
                fs'32
                g'8.
                r8 ~
                r32
            }
        }
    }
>>
```



Refer to the `BeatwiseQSchema` and `MeasurewiseQSchema` API entires for more information on controlling `Quantizer` output.

Refer to the `SearchTree` API entry for information on controlling the rhythmic complexity of `Quantizer` output.

3.1.4 The `timerelationtools` package

A new `timerelationtools` package is now available.

The `timerelationtools` package features seven functions for using natural language to compare the in-time position on an offset relative to a reference timespan:

```
timerelationtools.offset_happens_after_timespan_starts()
timerelationtools.offset_happens_after_timespan_stops()
timerelationtools.offset_happens_before_timespan_starts()
timerelationtools.offset_happens_before_timespan_stops()
timerelationtools.offset_happens_during_timespan()
timerelationtools.offset_happens_when_timespan_starts()
timerelationtools.offset_happens_when_timespan_stops()
```

The `timerelationtools` package contains `26` functions for using natural language to compare the in-time position of one timespan relative to another:

```
timerelationtools.timespan_2_contains_timespan_1_improperly()
timerelationtools.timespan_2_curtails_timespan_1()
timerelationtools.timespan_2_delays_timespan_1()
timerelationtools.timespan_2_happens_during_timespan_1()
timerelationtools.timespan_2_intersects_timespan_1()
timerelationtools.timespan_2_is_congruent_to_timespan_1()
timerelationtools.timespan_2_overlaps_all_of_timespan_1()
timerelationtools.timespan_2_overlaps_only_start_of_timespan_1()
timerelationtools.timespan_2_overlaps_only_stop_of_timespan_1()
timerelationtools.timespan_2_overlaps_start_of_timespan_1()
timerelationtools.timespan_2_overlaps_stop_of_timespan_1()
timerelationtools.timespan_2_starts_after_timespan_1_starts()
timerelationtools.timespan_2_starts_after_timespan_1_stops()
timerelationtools.timespan_2_starts_before_timespan_1_starts()
```

```
timerelationtools.timespan_2_starts_before_timespan_1_stops()
timerelationtools.timespan_2_starts_during_timespan_1()
timerelationtools.timespan_2_starts_when_timespan_1_starts()
timerelationtools.timespan_2_starts_when_timespan_1_stops()
timerelationtools.timespan_2_stops_after_timespan_1_starts()
timerelationtools.timespan_2_stops_after_timespan_1_stops()
timerelationtools.timespan_2_stops_before_timespan_1_starts()
timerelationtools.timespan_2_stops_before_timespan_1_stops()
timerelationtools.timespan_2_stops_during_timespan_1()
timerelationtools.timespan_2_stops_when_timespan_1_starts()
timerelationtools.timespan_2_stops_when_timespan_1_stops()
timerelationtools.timespan_2_trisects_timespan_1()
```

Here's an example of some of the natural language comparison functions available in the `timerelationtools` package:

```
>>> staff_1 = Staff(r"\times 2/3 { c'4 d'4 e'4 } \times 2/3 { f'4 g'4 a'4 }")
>>> staff_2 = Staff("c'2. d'4")
>>> score = Score([staff_1, staff_2])
```

```
>>> f(score)
\new Score <<
    \new Staff {
        \times 2/3 {
            c'4
            d'4
            e'4
        }
        \times 2/3 {
            f'4
            g'4
            a'4
        }
    }
    \new Staff {
        c'2.
        d'4
    }
>>
```

```
>>> last_tuplet = staff_1[-1]
>>> long_note = staff_2[0]
```

```
>>> timerelationtools.timespan_2_happens_during_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

```
>>> timerelationtools.timespan_2_intersects_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
True
```

```
>>> timerelationtools.timespan_2_is_congruent_to_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

```
>>> timerelationtools.timespan_2_overlaps_all_of_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

```
>>> timerelationtools.timespan_2_overlaps_start_of_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
True
```

```
>>> timerelationtools.timespan_2_overlaps_stop_of_timespan_1(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

```
>>> timerelationtools.timespan_2_starts_after_timespan_1_starts(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

```
>>> timerelationtools.timespan_2_starts_after_timespan_1_stops(
... timespan_1=last_tuplet, timespan_2=long_note)
False
```

3.1.5 Other new features

Autocompletion is now available at the Abjad prompt.

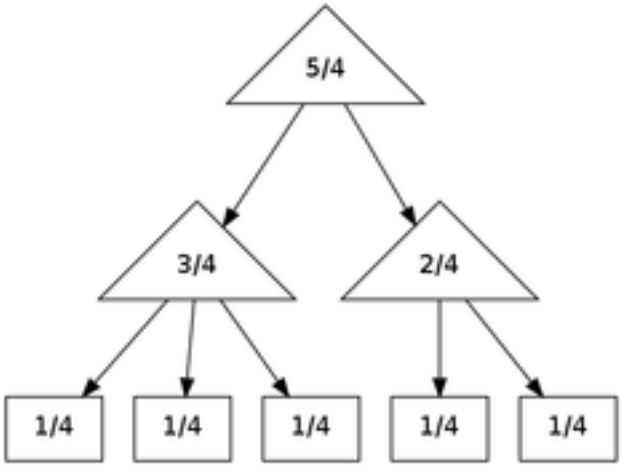
New tutorials describe how to get started with Abjad:

```
Getting started
LilyPond "hello, world!"
Python "hello, world!" (at the interpreter)
Python "hello, world!" (in a file)
More about Python
Abjad "hello, world" (at the interpreter)
Abjad "hello, world!" (in a file)
More about Abjad
```

Music notation images now appear in the docstrings of many functions throughout the API.

Added new `iotools.graph()` function to the `iotools` package. A small number of classes throughout the system have started to gain a *graphviz_format* attribute, including `datastructuretools.Digraph`, `documentationtools.InheritanceGraph`, some of the `rhythmtreetools.RhythmTreeNode` subclasses, and even `timesignaturetools.MetricalHierarchy`:

```
>>> hierarchy = timesignaturetools.MetricalHierarchy((7, 4))
>>> iotools.graph(hierarchy)
```



Forced accidentals and cautionary accidentals are now available as properties:

```
>>> note = notetools.Note("c'4")
>>> note.note_head.is_forced = True
>>> f(note)
c'!4
```

```
>>> note.note_head.is_cautionary = True
>>> f(note)
c'!?4
```

Forced accidentals and cautionary accidentals are also now available at instantiation:

```
>>> note = Note("c'!?4")
>>> note
Note("c'!?4")
```

```
>>> chord = Chord("<c'!? e'? g'! b'>4")
>>> chord
Chord("<c'!? e'? g'! b'>4")
```

```
>>> Note(chord)
Note("c'!?4")
```

```
>>> Chord(note)
Chord("<c'!?>4")
```

Added a function to register custom markup globally with the LilyPond parser:

```
>>> from abjad.tools.lilypondparsertools import LilyPondParser
```

```
>>> name = 'my-custom-markup-function'
>>> signature = ['markup?']
>>> LilyPondParser.register_markup_function(name, signature)
```

```
>>> parser = LilyPondParser()
>>> string = r"\markup { \my-custom-markup-function { foo bar baz } }"
>>> parser(string)
Markup((MarkupCommand('my-custom-markup-function', ['foo', 'bar', 'baz']),))
```

```
>>> f(_)
\markup { \my-custom-markup-function { foo bar baz } }
```

Note that this once registered, the custom markup command is also recognized when instantiating parsed markup objects:

```
>>> markuptools.Markup(r"\my-custom-markup-function { foo bar baz }")
Markup((MarkupCommand('my-custom-markup-function', ['foo', 'bar', 'baz']),))
```

Added new `markuptools.MusicGlyph` class. This is a subclass of `markuptools.MarkupCommand`, and can therefore be used anywhere `MarkupCommand` can appear. It guarantees correct quoting around the glyph name (which is easy to forget, or not always clear how to do for new users), and also checks that the glyph name is recognized in LilyPond:

```
>>> markuptools.MusicGlyph('accidentals.sharp')
MusicGlyph('accidentals.sharp')
```

```
>>> print _
\musicglyph #"accidentals.sharp"
```

The `durationtools` package now implements three related classes. All three classes are now available in the global namespace. Durations, multipliers and offsets are now distinguished everywhere in Abjad:

```
Duration
Multiplier
Offset
```

Implemented new `NonreducedRatio` class. Compare with existing `Ratio` class:

```
>>> mathtools.NonreducedRatio(2, 4, 2)
NonreducedRatio(2, 4, 2)
```

```
>>> mathtools.Ratio(2, 4, 2)
Ratio(1, 2, 1)
```

Added new `componenttools.ScoreSelection` subclasses. All selections are improper:

```
componenttools.Descendants
componenttools.Lineage
componenttools.Parentage
```

New score selection subclasses are also accessible via:

```
Component.descendants
Component.lineage
Component.parentage
```

Added `lilypondfiletools.LilyPondDimension` class:

```
>>> dimension = lilypondfiletools.LilyPondDimension(2, 'in')
>>> f(dimension)
2.0\in
```

Added a new parseable tag to abjad-book: `<abjadextract module \>[flags]`. This single-line tag imports the code found at *module*, and copies the actual code text itself into the abjad-book session, just as though it had been manually included between a pair of `<abjad></abjad>` tags. The intended use is in Abjad's literature examples. Most of the examples are also written up in the `demos/` directory.

The `abjad-book` executable now handles multi-page PNG output.

Implemented page selection in `abjad-book`. Example: show pages 2 through 5 of a multipage score:

```
<abjad>
show(foo) <page 2-5
</abjad>
```

Added new `EvenRunRhythmMaker` class to the `rhythmmakertools` package. For each division on which the class is called, the class produces an even run of notes each equal in duration to `1/d` with `d` equal to the division denominator:

```
>>> maker = rhythmmakertools.EvenRunRhythmMaker()
```

```
>>> divisions = [(4, 16), (3, 8), (2, 8)]
>>> lists = maker(divisions)
>>> containers = sequencetools.flatten_sequence(lists)
```

```
>>> staff = Staff(containers)
```

```
>>> f(staff)
\new Staff {
    {
        c'16 [
        c'16
        c'16
        c'16 ]
    }
    {
        c'8 [
        c'8
        c'8 ]
    }
    {
        c'8 [
        c'8 ]
    }
}
```

3.2 Older versions

3.2.1 Abjad 2.10

Released 2012-10-05. Built from r7615. Implements 437 public classes and 982 functions totalling 179,000 lines of code.

The following packages now load by default when you start Abjad:

```
Abjad 2.10
>>> [x for x in dir() if x.endswith('tools')]
['abjadbooktools', 'beamtools', 'chordtools', 'componenttools', 'containertools', 'contexttools',
'developerscripttools', 'durationtools', 'formattools', 'gracetools', 'instrumenttools',
'introspectiontools', 'iotools', 'iterationtools', 'labeltools', 'layouttools', 'leaftools',
'lilypondfiletools', 'marktools', 'markuptools', 'mathtools', 'measuretools', 'notetools', '
'offsettools', 'pitcharraytools', 'pitchtools', 'resttools', 'rhythmtreetools', 'schemetools',
```

```
'scoretemplatetools', 'scoretools', 'sequencetools', 'sievetools', 'skiptools', 'spannertools',
'stafftools', 'stringtools', 'tempotools', 'tietools', 'timeintervaltools', 'timesignaturetools',
'rhythmmakertools', 'tonalitytools', 'tuplettools', 'verticalitytools', 'voicetools']
```

Improved formatting engine. Scores now format approximately 30% faster.

Improved LilyPond parser.

Markup objects now parse input string input on initialization:

```
>>> markuptools.Markup(r'\bold \tiny { foo bar baz }')
Markup((MarkupCommand('bold', MarkupCommand('tiny', ['foo', 'bar', 'baz'])),))
```

```
>>> print _.indented_lilypond_format
\markup {
    \bold
        \tiny
            {
                foo
                bar
                baz
            }
    }
```

You can now use context names to reference named contexts attached to any container:

```
>>> template = scoretemplatetools.StringQuartetScoreTemplate()
>>> score = template()
```

```
>>> score['First Violin Staff']
Staff-"First Violin Staff"{1}
```

```
>>> score['First Violin Voice']
Voice-"First Violin Voice"{}
```

Five new constants are available globally.

- The constants are `Left`, `Right`, `Up`, `Down` and `Center`.
- The constants function like Python's built-in `True` and `False`.
- Use the constants as keyword defaults.

A new configuration tool is available:

```
configurationtools.get_abjad_startup_string()
```

New context tools are available:

```
contexttools.all_are_contexts()
```

A new `iterationtools` package is available:

```
iterationtools.iterate_chords_in_expr()
iterationtools.iterate_components_and_grace_containers_in_expr()
iterationtools.iterate_components_depth_first()
iterationtools.iterate_components_in_expr()
iterationtools.iterate_containers_in_expr()
iterationtools.iterate_contexts_in_expr()
iterationtools.iterate_leaf_pairs_in_expr()
iterationtools.iterate_leaves_in_expr()
iterationtools.iterate_measures_in_expr()
iterationtools.iterate_namesakes_from_component()
iterationtools.iterate_notes_and_chords_in_expr()
iterationtools.iterate_notes_in_expr()
iterationtools.iterate_rests_in_expr()
iterationtools.iterate_scores_in_expr()
iterationtools.iterate_semantic_voices_in_expr()
iterationtools.iterate_skips_in_expr()
iterationtools.iterate_staves_in_expr()
iterationtools.iterate_thread_from_component()
```

```
iterationtools.iterate_thread_in_expr()
iterationtools.iterate_timeline_from_component()
iterationtools.iterate_timeline_in_expr()
iterationtools.iterate_tuplets_in_expr()
iterationtools.iterate_voices_in_expr()
```

New LilyPond file tools are available:

```
lilypondfiletools.make_floating_time_signature_lilypond_file()
```

New LilyPond parser tools are available:

```
lilypondparsertools.GuileProxy
lilypondparsertools.LilyPondDuration
lilypondparsertools.LilyPondEvent
lilypondparsertools.LilyPondFraction
lilypondparsertools.LilyPondLexicalDefinition
lilypondparsertools.LilyPondSyntacticalDefinition
lilypondparsertools.ReducedLyParser
lilypondparsertools.SchemeParser
lilypondparsertools.SyntaxNode
lilypondparsertools.lilypond_enharmonic_transpose()
```

A new `Ratio` class is available in the `mathtools` package:

```
>>> mathtools.Ratio(1, 2, -1)
Ratio(1, 2, -1)
```

New rhythm-tree tools are available.

- Implemented RTM expression parser:

```
rhythmtreetools.RhythmTreeParser
```

- Implemented new classes for explicitly constructing rhythm-trees:

```
RhythmTreeNode
RhythmTreeLeaf
RhythmTreeContainer
```

```
>>> from abjad import *
>>> rtm = '(1 (1 (2 (1 -1 1)) -2))'
>>> result = rhythmtreetools.RhythmTreeParser()(rtm)
```

```
>>> result[0]
RhythmTreeContainer(
    children=(
        RhythmTreeLeaf(
            duration=1,
            pitched=True,
            ),
        RhythmTreeContainer(
            children=(
                RhythmTreeLeaf(
                    duration=1,
                    pitched=True,
                    ),
                RhythmTreeLeaf(
                    duration=1,
                    pitched=False,
                    ),
                RhythmTreeLeaf(
                    duration=1,
                    pitched=True,
                    ),
            ),
            duration=2
            ),
        RhythmTreeLeaf(
            duration=2,
            pitched=False,
```

```
                ),
        ),
        duration=1
        )
```

```
>>> _.rtm_format
'(1 (1 (2 (1 -1 1)) -2))'
```

```
>>> result[0]((1, 4))
FixedDurationTuplet(1/4, [c'16, {@ 3:2 c'16, r16, c'16 @}, r8])
```

```
>>> f(_)
\times 4/5 {
    c'16
    \times 2/3 {
        c'16
        r16
        c'16
    }
    r8
}
```

New Scheme tools are available.

- Added `force_quotes` boolean keyword to `schemetools.Scheme` and `schemetools.format_scheme_value()`:

```
>>> schemetools.format_scheme_value('foo')
'foo'
```

```
>>> schemetools.format_scheme_value('foo', force_quotes=True)
'"foo"'
```

  This allows you to force double quotes around strings which contain no spaces. This is necessary for some LilyPond grob overrides.

- A new Scheme formatting function is available:

```
schemetools.format_scheme_value()
```

New score-template tools are available:

```
scoretemplatetools.GroupedStavesScoreTemplate
```

New sequence tools are available:

- Added `sequencetools.merge_duration_sequences()`:

```
>>> sequencetools.merge_duration_sequences([10, 10, 10], [7])
[7, 3, 10, 10]
```

- Added `sequencetools.pair_duration_sequence_elements_with_input_pair_values()`:

```
>>> duration_sequence = [10, 10, 10, 10]
>>> input_pairs = [('red', 1), ('orange', 18), ('yellow', 200)]
>>> sequencetools.pair_duration_sequence_elements_with_input_pair_values(
... duration_sequence, input_pairs)
[(10, 'red'), (10, 'orange'), (10, 'yellow'), (10, 'yellow')]
```

New tie tools are available:

```
tietools.get_tie_spanner_attached_to_component()
```

New time-interval tools are available:

```
timeintervaltools.make_voice_from_nonoverlapping_intervals()
```

New time-token tools are available:

- Added `SkipRhythmMaker` to `rhythmmakertools` package:

```
>>> maker = rhythmmakertools.SkipRhythmMaker()
```

```
>>> duration_tokens = [(1, 5), (1, 4), (1, 6), (7, 9)]
>>> leaf_lists = maker(duration_tokens)
>>> leaves = sequencetools.flatten_sequence(leaf_lists)
```

```
>>> staff = Staff(leaves)
```

```
>>> f(staff)
\new Staff {
    s1 * 1/5
    s1 * 1/4
    s1 * 1/6
    s1 * 7/9
}
```

- Added `TupletMonadRhythmMaker` to `rhythmmakertools` package:

```
>>> maker = rhythmmakertools.TupletMonadRhythmMaker()
```

```
>>> duration_tokens = [(1, 5), (1, 4), (1, 6), (7, 9)]
>>> tuplets = maker(duration_tokens)
>>> staff = Staff(tuplets)
```

```
>>> f(staff)
\new Staff {
    \times 4/5 {
        c'4
    }
    {
        c'4
    }
    \times 2/3 {
        c'4
    }
    \times 8/9 {
        c'2..
    }
}
```

3.2.2 Abjad 2.9

Released 2012-06-05. Built from r5795. Implements 405 public classes and 1066 functions totalling 182,000 lines of code.

Extended markup handling is now available.

- The LilyPond parser accepts complex markup as input:

```
>>> f(p(r'''{ c'4 _ \markup { \put-adjacent #1 #-1 \bold \fontsize #2 \upright foo bar } }'''))
{
    c'4
        _ \markup {
            \put-adjacent
                #1
                #-1
                \bold
                    \fontsize
                        #2
                        \upright
                            foo
                bar
            }
}
```

- Format routines allow for markup indentation:

```
>>> circle = markuptools.MarkupCommand('draw-circle', 2.5, 0.1, False)
>>> square = markuptools.MarkupCommand('rounded-box', 'hello?')
>>> line = markuptools.MarkupCommand('line', [square, 'wow!'])
>>> markup = markuptools.Markup(('X', square, 'Y', line, 'Z'), direction='up')
```

```
>>> print '\n'.join(markup._get_format_pieces(is_indented=True))
^ \markup {
    X
    \rounded-box
        hello?
    Y
    \line
        {
            \rounded-box
                hello?
            wow!
        }
    Z
    }
```

- Nontrivial markup format with indentation automatically:

```
>>> staff = Staff("c")
>>> m1 = markuptools.Markup('foo')(staff[0])
>>> m2 = markuptools.Markup('bar')(staff[0])
>>> m3 = markuptools.Markup('baz', 'up')(staff[0])
>>> m4 = markuptools.Markup('quux', 'down')(staff[0])
>>> accent = marktools.Articulation('accent')(staff[0])
```

```
>>> f(staff)
\new Staff {
    c4 -\accent
        ^ \markup { baz }
        _ \markup { quux }
        - \markup {
            \column
                {
                    foo
                    bar
                }
            }
}
```

- `Markup.contents` is now a tuple of strings or `MarkupCommand` instances.
- Removed the markup `style_string` property. Use `schemetools` classes for constructing Scheme-style formatting.
- Changed `Markup.contents_string` to `Markup.contents`.

An entirely new tuplet microlanguage is now available.

- This "reduced ly" syntax uses braces to show tuplet nesting and represents rhythm without pitch:

```
>>> from abjad.tools import rhythmtreetools
```

```
>>> container = rhythmtreetools.parse_reduced_ly_syntax('4 -4 8 5/3 { 2/3 { 8 8 8 } { 8 8 } -8 } 4')
```

```
>>> f(container)
{
    c'4
    r4
    c'8
    \fraction \times 5/3 {
        \times 2/3 {
            c'8
            c'8
            c'8
        }
        {
            c'8
```

```
                c'8
            }
            r8
        }
        c'4
    }
```

- Measures and dotted values are also available:

```
>>> container = rhythmtreetools.parse_reduced_ly_syntax('|2/4 8. 16 8. 16| |4/4 2/3 { 2 2 2 }|')
```

```
f(container)
```

```
{
    {
        \time 2/4
        c'8.
        c'16
        c'8.
        c'16
    }
    {
        \time 4/4
        \times 2/3 {
            c'2
            c'2
            c'2
        }
    }
}
```

Extended container input syntax.

- You can now pass strings directly to the `append()` and `extend()` methods of any container:

```
>>> container = Container()
>>> container
{}
```

```
>>> container.extend('a b c')
>>> container
{a4, b4, c4}
```

```
>>> container.append('d')
>>> container
{a4, b4, c4, d4}
```

- You can assign a string to any container item:

```
>>> container = Container("c' d' e'")
>>> container
{c'4, d'4, e'4}
```

```
>>> container[1] = 'r'
>>> container
{c'4, r4, e'4}
```

- You can assign a string to any container slice:

```
>>> container = Container("c' d' e'")
>>> container
{c'4, d'4, e'4}
```

```
>>> container[:2] = 'r8 r r'
>>> container
{r8, r8, r8, e'4}
```

- You can initialize containers from strings using alternate parsers.

Use the `'abj'` prefix to initialize a container with the new reduced ly syntax:

```
>>> staff = Staff('abj: | 2/4 2/3 { 8 4 } 8 8 || 3/4 4 4 4 |')
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/4
        \times 2/3 {
            c'8
            c'4
        }
        c'8
        c'8
    }
    {
        \time 3/4
        c'4
        c'4
        c'4
    }
}
```

- Use the `'rtm'` prefix to initialize a container with IRCAM RTM-style syntax:

```
>>> staff = Staff('rtm: (1 (1 (2 (1 1 1)) 1)) (1 (1 1))')
```

```
>>> f(staff)
\new Staff {
    c'16
    \times 2/3 {
        c'16
        c'16
        c'16
    }
    c'16
    c'8
    c'8
}
```

- Parallel contexts, such as Score, can be instantiated from strings which parse to a sequence of contexts:

```
Score(r'''\new Staff { c' } \new Staff = { c, }''')
```

- Added a new `FixedDurationContainer` class to the `containertools` package.

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

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

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

```
>>> f(container)
{
    c'8
    d'8
    e'8
}
```

```
>>> container.is_misfilled
False
```

```
>>> container.pop()
Note("e'8")
```

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

```
>>> container.is_misfilled
True
```

Misfilled fixed-duration containers will raise an exception at format-time. Fixed-duration containers share this behavior with measures.

Regularized measure modification behavior.

- By default measures do not automatically adjust time signature after contents modification:

```
>>> measure = Measure((3, 4), "c' d' e'")
>>> measure
Measure(3/4, [c'4, d'4, e'4])
```

```
>>> measure.append('r')
>>> measure
Measure(3/4, [c'4, d'4, e'4, r4])
```

```
>>> measure.is_overfull
True
```

- But it is now possible to cause measures to automatically adjust time signature after contents modification:

```
>>> measure = Measure((3, 4), "c' d' e'")
>>> measure.automatically_adjust_time_signature = True
>>> measure
Measure(3/4, [c'4, d'4, e'4])
```

```
>>> measure.append('r')
>>> measure
Measure(4/4, [c'4, d'4, e'4, r4])
```

```
>>> measure.is_misfilled
False
```

Previous implementations of measure `append()`, `extend()` and set-item never adjusted measure time signatures.

Now the behavior of such operations is controllable on a measure-by-measure basis by the end user.

New functionality is available for working with ties.

- Added a `TieChain` class to the `tietools` package. Tie chains now return as a custom `TieChain` object instead of tuple:

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

```
>>> tietools.get_tie_chain(staff[2])
TieChain((Note("e'4"), Note("e'4")))
```

Reimplemented tie chain duration attributes as explicit class attributes. The following four functions have been removed:

```
tietools.get_preprolated_tie_chain_duration()
tietools.get_prolated_tie_chain_duration()
tietools.get_tie_chain_duration_in_seconds()
tietools.get_written_tie_chain_duration()
```

Use these read-only properties instead:

```
TieChain.preprolated_duration
TieChain.prolated_duration
TieChain.duration_in_seconds
TieChain.written_duration
```

The `TieChain` class inherits from the new `Selection` abstract base class.

Added new `tietools` functions:

```
tietools.iterate_pitched_tie_chains_forward_in_expr()
tietools.iterate_pitched_tie_chains_backward_in_expr()
tietools.iterate_nontrivial_tie_chains_forward_in_expr()
tietools.iterate_nontrivial_tie_chains_backward_in_expr()
```

Removed `tietools.is_tie_chain(expr)`. Use `isinstance(expr, tietools.TieChain)` instead.

Removed `tietools.get_leaves_in_tie_chain()`. Use `TieChain.leaves` instead.

Removed `tietools.group_leaves_in_tie_chain_by_immediate_parents()`. Use `TieChain.leaves_grouped_by_immediate_parents` instead.

Removed `tietools.is_tie_chain_with_all_leaves_in_same_parent()`. Use `TieChain.all_leaves_are_in_same_parent` instead.

Added a new `stringtools` package.

- The following functions all migrated from the `iotools` package:

```
stringtools.capitalize_string_start()
stringtools.format_input_lines_as_doc_string()
stringtools.format_input_lines_as_regression_test()
stringtools.is_lowercamelcase_string()
stringtools.is_space_delimited_lowercase_string()
stringtools.is_underscore_delimited_lowercase_file_name()
stringtools.is_underscore_delimited_lowercase_file_name_with_extension()
stringtools.is_underscore_delimited_lowercase_package_name()
stringtools.is_underscore_delimited_lowercase_string()
stringtools.is_uppercamelcase_string()
stringtools.space_delimited_lowercase_to_uppercamelcase()
stringtools.string_to_strict_directory_name()
stringtools.strip_diacritics_from_binary_string()
stringtools.underscore_delimited_lowercase_to_lowercamelcase()
stringtools.underscore_delimited_lowercase_to_uppercamelcase()
stringtools.uppercamelcase_to_space_delimited_lowercase()
stringtools.uppercamelcase_to_underscore_delimited_lowercase()
```

The package also contains these new functions:

```
stringtools.arg_to_bidirectional_direction_string()
stringtools.arg_to_bidirectional_lilypond_symbol()
stringtools.arg_to_tridirectional_direction_string()
stringtools.arg_to_tridirectional_lilypond_symbol()
```

```
>>> stringtools.arg_to_bidirectional_lilypond_symbol(1)
'^'
>>> stringtools.arg_to_tridirectional_direction_string('-')
'neutral'
```

Added a new `beamtools` package.

- This release of the `beamtools` package contains the following classes and functions:

```
beamtools.BeamSpanner
beamtools.ComplexBeamSpanner
beamtools.DuratedComplexBeamSpanner
beamtools.MultipartBeamSpanner
```

```
beamtools.is_beamable_component
beamtools.apply_beam_spanner_to_measure
beamtools.apply_beam_spanners_to_measures_in_expr
beamtools.apply_complex_beam_spanner_to_measure
beamtools.apply_complex_beam_spanners_to_measures_in_expr
beamtools.apply_durated_complex_beam_spanner_to_measures
beamtools.beam_bottommost_tuplets_in_expr
beamtools.get_beam_spanner_attached_to_component
beamtools.is_beamable_component
beamtools.is_component_with_beam_spanner_attached
```

Note that the following two functions have been removed:

```
beamtools.apply_beam_spanner_to_measure()
beamtools.apply_complex_beam_spanner_to_measure()
```

Use these two functions instead:

```
beamtools.apply_beam_spanners_to_measures_in_expr()
beamtools.apply_complex_beam_spanners_to_measures_in_expr()
```

New `constrainttools` functionality is now available.

- Extended the `VariableLengthStreamSolver` class.

  The class now produces more randomly ordered solution sets than before, when in randomized mode. Note that the solution sets tend to increase in size. Also note that there is an increased performance hit for such PMC-style randomized constraint solving:

```
>>> from abjad.tools.constrainttools import *
```

```
>>> domain = Domain([1, 2, 3, 4], 1)
>>> boundary_sum = GlobalConstraint(lambda x: sum(x) < 6)
>>> target_sum = GlobalConstraint(lambda x: sum(x) == 5)
>>> random_solver = VariableLengthStreamSolver(domain,
... [boundary_sum], [target_sum], randomized=True)
>>> for x in random_solver: x
...
[1, 3, 1]
[4, 1]
[3, 2]
[2, 3]
[1, 4]
[3, 1, 1]
[2, 1, 2]
[1, 2, 1, 1]
[2, 1, 1, 1]
[2, 2, 1]
[1, 1, 1, 2]
[1, 2, 2]
[1, 1, 1, 1, 1]
[1, 1, 3]
[1, 1, 2, 1]
```

- Randomized the `FixedLengthStreamSolvers` class.

  The class now produces truly randomly ordered solution sets.

New sequence tools are available.

- Added new type- and form-checking predicates to the `sequencetools` package:

```
sequencetools.all_are_integer_equivalent_exprs
sequencetools.is_null_tuple(expr)
sequencetools.is_singleton(expr)
sequencetools.is_pair(expr)
sequencetools.is_n_tuple(expr, n)
sequencetools.is_integer_singleton(expr)
sequencetools.is_integer_pair(expr)
sequencetools.is_integer_n_tuple(expr, n)
sequencetools.is_integer_equivalent_n_tuple
sequencetools.is_integer_equivalent_pair
sequencetools.is_integer_equivalent_singleton
sequencetools.is_fraction_equivalent_pair
```

Each function returns a boolean:

```
>>> sequencetools.is_integer_singleton((19,))
True
```

- Added a new `NonreducedFraction` class to the `sequencetools` package:

```
>>> sequencetools.NonreducedFraction(3, 6)
NonreducedFraction(3, 6)
```

Like built-in fraction but numerator and denominator do NOT simplify.

All six comparators are implemented on nonreduced fractions.

Addition and subtraction are implemented on nonreduced fractions:

```
>>> sequencetools.NonreducedFraction(3, 6) + sequencetools.NonreducedFraction(3, 6)
NonreducedFraction(6, 6)
```

Use nonreduced fractions to model arithmetic operations on time signature-like objects absent any of the special time signature features like partial-measure pick-ups.

New spanners and spanner handlers are now available.

- Added a `ComplexGlissandoSpanner` to the `spannertools` package.

  This spanner generates a glissando which skips over rests. It can be used in combination with spannertools.BeamSpanner and an override of the Stem grob to generate the appearance of durated glissandi:

```
>>> staff = Staff("c'16 [ d' r e' r r r g' ]")
```

```
>>> f(staff)
\new Staff {
    c'16 [
    d'16
    r16
    e'16
    r16
    r16
    r16
    g'16 ]
}

>>> spannertools.ComplexGlissandoSpanner(staff[:])
ComplexGlissandoSpanner(c'16, d'16, r16, e'16, r16, r16, r16, g'16)
```

```
>>> staff.override.stem.stemlet_length = 2
>>> f(staff)
\new Staff \with {
    \override Stem #'stemlet-length = #2
} {
    c'16 [ \glissando
    d'16 \glissando
    \once \override NoteColumn #'glissando-skip = ##t
    \once \override Rest #'transparent = ##t
    r16
    e'16 \glissando
    \once \override NoteColumn #'glissando-skip = ##t
    \once \override Rest #'transparent = ##t
    r16
    \once \override NoteColumn #'glissando-skip = ##t
    \once \override Rest #'transparent = ##t
    r16
    \once \override NoteColumn #'glissando-skip = ##t
    \once \override Rest #'transparent = ##t
    r16
    g'16 ]
}
```

- Added new `spannertools` function:

```
spannertools.destory_spanners_attached_to_components_in_expr(expr, klass=None)
```

  The function can be useful for removing all spanners when debugging a complex expression.

- Spanners are now callable:

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

```
>>> beam = spannertools.BeamSpanner()
>>> beam(staff[:])
Staff{4}
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8
    e'8
    f'8 ]
}
```

This works the same way as marks:

```
>>> marktools.Articulation('.')(staff[1])
Articulation('.')(d'8)
```

```
>>> f(staff)
\new Staff {
    c'8 [
    d'8 -\staccato
    e'8
    f'8 ]
}
```

Callable spanners are provided as an experimental way of unifying the attachment syntax of spanners and marks.

Many new functions are available in the `componenttools` package.

- New getters:

```
componenttools.get_proper_contents_of_component()
componenttools.get_improper_contents_of_component()
componenttools.get_improper_contents_of_component_that_start_with_component()
componenttools.get_improper_contents_of_component_that_stop_with_component()
componenttools.get_proper_descendents_of_component()
componenttools.get_improper_descendents_of_component()
componenttools.get_improper_descendents_of_component_that_cross_prolated_offset
componenttools.get_improper_descendents_of_component_that_start_with_component
componenttools.get_improper_descendents_of_component_that_stop_with_component
componenttools.get_lineage_of_component()
componenttools.get_lineage_of_component_that_start_with_component()
componenttools.get_lineage_of_component_that_stop_with_component()
componenttools.get_nth_sibling_from_component(component, n)
componenttools.get_nth_component_from_component_in_time_order(component, n)
componenttools.get_nth_namesake_from_component
componenttools.get_most_distant_sequential_container_in_improper_parentage_of_component()
```

Use these functions to interrogate the structural relations of components resident inside arbitrarily complex pieces of score.

The functions are useful as primitive methods when implementing more complex operations designed to mutate the score tree.

- Note the difference between the 'contents' of a component and the 'descendents' of a component:

```
>>> componenttools.get_proper_contents_of_component(staff)
[Note("c'4"), Tuplet(2/3, [d'8, e'8, f'8])]
```

Versus:

```
>>> componenttools.get_proper_descendents_of_component(staff)
[Note("c'4"), Tuplet(2/3, [d'8, e'8, f'8]), Note("d'8"), Note("e'8"), Note("f'8")]
```

- Also add the following `componenttools` predicate:

```
componenttools.is_immediate_temporal_successor_of_component()
```

Further new functionality:

- Added new `gracetools` function:

```
gracetools.detach_grace_containers_attached_to_leaves_in_expr()
```

Use the function to strip all grace containers from an arbitrary piece of score.

- Added new `marktools` functions:

```
marktools.get_marks_attached_to_components_in_expr()
marktools.detach_marks_attached_to_components_in_expr()
marktools.move_marks(donor, recipient).
```

- Added new `pitchtools` function:

```
pitchtools.set_written_pitch_of_pitched_components_in_expr(expr, written_pitch=0)
```

Use the function to neutralize pitch information in an arbitrary piece of score.

- Added new `tuplettools` functions:

```
tuplettools.change_fixed_duration_tuplets_in_expr_to_tuplets()
tuplettools.change_tuplets_in_expr_to_fixed_duration_tuplets()
```

- Extended `lilypondfiletools.ContextBlock` with the following attributes:

```
ContextBlock.engraver_consists
ContextBlock.engraver_removals
ContextBlock.context_name
ContextBlock.name
ContextBlock.type
```

The attributes correspond to backslash-initiated LilyPond commands available in LilyPond context blocks.

- Updated `LilyPondLanguageToken` to format LilyPond `\language` command instead of LilyPond `\include` command.
- Extended `Duration` to initialize from LilyPond duration strings:

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

Note that this means that `Duration('2')` now gives `Duration(1, 2)`. Previously `Duration('2')` gave `Duration(2, 1)` just like `Fraction('2')`.

Changes to end-user functionality:

- Changed:

```
componenttools.copy_components_and_remove_all_spanners()
```

```
componenttools.copy_components_and_remove_spanners()
```

- Changed:

```
componenttools.get_improper_contents_of_component_that_cross_prolated_offset()
```

```
componenttools.get_leftmost_components_with_total_duration_at_most()
```

- Changed:

```
componenttools.list_improper_contents_of_component_that_cross_prolated_offset()
```

```
componenttools.list_leftmost_components_with_prolated_duration_at_most()
```

- Changed:

```
configurationtool.set_default_accidental_spelling()
```

```
pitchtools.set_default_accidental_spelling()
```

- Changed:

```
gracetools.Grace
```

```
gracetools.GraceContainer
```

- Changed:

```
spannertools.destory_all_spanners_attached_to_component()
```

```
spannertools.destory_spanners_attached_to_component()
```

- Changed:

```
spannertools.fracture_all_spanners_attached_to_component()
```

```
spannertools.fracture_spanners_attached_to_component()
```

- Changed:

```
spannertools.report_as_string_format_contributions_of_all_spanners_attached_to_component()
```

```
spannertools.report_as_string_format_contributions_of_spanners_attached_to_component()
```

- Changed:

```
spannertools.report_as_string_format_contributions_of_all_spanners_attached_to_improper_parentage_of_com
```

```
spannertools.report_as_string_format_contributions_of_spanners_attached_to_improper_parentage_of_compone
```

- Changed:

```
tietools.get_tie_chains_in_expr()
```

```
tietools.get_nontrivial_tie_chains_masked_by_components()
```

- Changed:

```
tietools.remove_all_leaves_in_tie_chain_except_first()
```

```
tietools.remove_nonfirst_leaves_in_tie_chain()
```

- Changed:

```
scr/devel/rename-public-helper
```

```
scr/devel/rename-public-function
```

- Removed the `threadtools` package and moved all functions to `componenttools`.

  Instead of these:

```
threadtools.iterate_thread_backward_from_component()
threadtools.iterate_thread_backward_in_expr()
threadtools.iterate_thread_forward_from_component()
threadtools.iterate_thread_forward_in_expr()
threadtools.component_to_thread_signature()
```

  Use these:

```
componenttools.iterate_thread_backward_from_component()
componenttools.iterate_thread_backward_in_expr()
componenttools.iterate_thread_forward_from_component()
componenttools.iterate_thread_forward_in_expr()
componenttools.component_to_containment_signature()
```

- Removed the read-only `Component.marks` property entirely.
- Removed the top-level `abjad/exceptions` directory. Use the new `exceptiontools` package instead.

- Removed the top-level `abjad/templates` directory.

  Make sure to read the changes carefully.

  If you have been working with grace notes, for example, you will need to change all occurrences of `gracetools.Grace` to `gracetools.GraceContainer`.

3.2.3 Abjad 2.8

Released 2012-04-16. Built from r5421. Implements 306 public classes and 1037 functions totalling 178,000 lines of code.

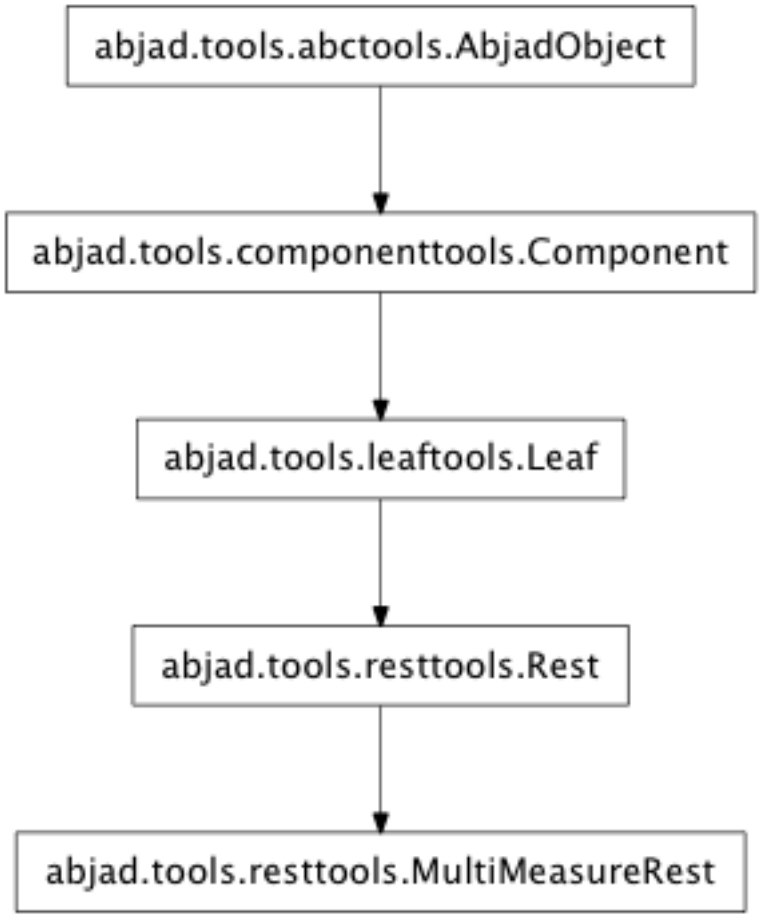
Many documentation improvements appear in this release.

- A source link now accompanies all classes and functions in the API:

Source code for abjad.tools.chordtools.arpeggiate_chord

```
from abjad.tools.chordtools.Chord import Chord
from abjad.tools.decoratortools import requires


@requires(Chord)
def arpeggiate_chord(chord):                                                    [docs]
    '''.. versionadded:: 1.1

    Arpeggiate `chord`::

        abjad> chord = Chord("<c' d'' ef''>8")

    ::

        abjad> chordtools.arpeggiate_chord(chord)
        [Note("c'8"), Note("d''8"), Note("ef''8")]

    Arpeggiated notes inherit `chord` written duration.

    Arpeggiated notes do not inherit other `chord` attributes.

    Return list of newly constructed notes.

    .. versionchanged:: 2.0
        renamed ``chordtools.arpeggiate()`` to
        ``chordtools.arpeggiate_chord()``.
    '''
    from abjad.tools.notetools.Note import Note

    result = []
    chord_written_duration = chord.written_duration
    for pitch in chord.written_pitches:
        result.append(Note(pitch, chord_written_duration))

    return result
```

- All parts of the Abjad codebase are now viewable through the HTML version of the API.
- Inheritance diagrams now accompany all classes:



- Inherited attributes now appear in the API entry of each class.
- Added new `documentationtools` package:

```
documentationtools.APICrawler
documentationtools.AbjadAPIGenerator
documentationtools.ClassCrawler
documentationtools.ClassDocumenter
documentationtools.Documenter
documentationtools.FunctionCrawler
documentationtools.FunctionDocumenter
documentationtools.InheritanceGraph
documentationtools.ModuleCrawler
documentationtools.Pipe
```

The package houses custom code to build Abjad documentation.

Added the new `constrainttools` API.

- This release of the `constrainttools` package implements the following classes:

```
constrainttools.AbsoluteIndexConstraint
constrainttools.Domain
constrainttools.FixedLengthStreamSolver
constrainttools.GlobalConstraint
constrainttools.GlobalCountsConstraint
constrainttools.GlobalReferenceConstraint
constrainttools.RelativeCountsConstraint
constrainttools.RelativeIndexConstraint
constrainttools.VariableLengthStreamSolver
```

- Example:

```
>>> from abjad.tools.constraintstools import *
```

```
>>> domain = Domain([1, 2, 3, 4], 4)
```

```
>>> all_unique = GlobalCountsConstraint(lambda x: all([y == 1 for y in x.values()]))
>>> max_interval = RelativeIndexConstraint([0, 1], lambda x, y: abs(x - y) < 3)
>>> solver = FiniteStreamSolver(domain, [all_unique, max_interval])
```

```
>>> for solution in solver: print solution
...
(1, 2, 3, 4)
(1, 2, 4, 3)
(1, 3, 2, 4)
(1, 3, 4, 2)
(2, 1, 3, 4)
(2, 4, 3, 1)
(3, 1, 2, 4)
(3, 4, 2, 1)
(4, 2, 1, 3)
(4, 2, 3, 1)
(4, 3, 1, 2)
(4, 3, 2, 1)
```

- The `constrainttools` package is considered unstable and will be subject to changes in the next releases of Abjad.

Added octave-transposition mapping model.

- This version of the system contains the following classes:

```
pitchtools.OctaveTranspositionMapping
pitchtools.OctaveTranspositionMappingComponent
pitchtools.OctaveTranspositionMappingInventory
```

- Octave-transposition mappings specify a way to maybe pitches from one registral space to another.
- Use octave-transposition mappings as input to `pitchtools.transpose_chromatic_pitch_number_ty_octave`

Many Abjad classes are now implemented as abstract base classes.

- Abstract base classes provide functionality to child subclasses.
- Abstract base classes can not be instantiated directly.
- The Abjad API now lists abstract classes and concrete classes separately.
- See http://docs.python.org/library/abc.html for a description of ABCs in Python.

Added the new `abctools` package to house abstract classes that are core to the Abjad object model.

- This version of the package contains the following classes:

```
abctools.AbjadObject
abctools.AttributeEqualityAbjadObject
abctools.ImmutableAbjadObject
abctools.SortableAttributeEqualityAbjadObject
```

- All Abjad classes now inherit from `AbjadObject`.

Added object inventories for several classes.

- This release contains inventories for the following classes:

```
contexttools.ClefMarkInventory
contexttools.TempoMarkInventory
instrumenttools.InstrumentInventory
markuptools.MarkupInventory
pitchtools.OctaveTranspositionMappingInventory
pitchtools.PitchRangeInventory
scoretools.PerformerInventory
```

- Object inventories model ordered collections of system objects.

Add the new `datastructuretools` package.

- This version of the package includes the following classes:

```
datastructuretools.Digraph
datastructuretools.ImmutableDictionary
datastructuretools.ObjectInventory
```

- Use `datastructuretools.Digraph` to detect cycles in any collection of hashable objects:

```
>>> from abjad.tools.datastructuretools import Digraph
```

```
>>> edges = [('a', 'b'), ('a', 'c'), ('a', 'f'), ('c', 'd'), ('d', 'e'), ('e', 'c')]
>>> digraph = Digraph(edges)
>>> digraph
Digraph(edges=[('a', 'c'), ('a', 'b'), ('a', 'f'), ('c', 'd'), ('d', 'e'), ('e', 'c')])
```

```
>>> digraph.root_nodes
('a',)
>>> digraph.terminal_nodes
('b', 'f')
>>> digraph.cyclic_nodes
('c', 'd', 'e')
>>> digraph.is_cyclic
True
```

- Use `datastructuretools.ObjectInventory` as the base class for an ordered collection of system objects.
- Object inventories inherit from `list` and are mutable.
- Object inventories extend `append()`, `extend()` and `__contains__()` to allow token input.

Added new `wellformednesstools` package.

- This version of the package implements the following classes:

```
wellformednesstools.BeamedQuarterNoteCheck
wellformednesstools.DiscontiguousSpannerCheck
wellformednesstools.DuplicateIdCheck
wellformednesstools.EmptyContainerCheck
wellformednesstools.IntermarkedHairpinCheck
wellformednesstools.MisduratedMeasureCheck
wellformednesstools.MisfilledMeasureCheck
wellformednesstools.MispitchedTieCheck
wellformednesstools.MisrepresentedFlagCheck
wellformednesstools.MissingParentCheck
wellformednesstools.NestedMeasureCheck
wellformednesstools.OverlappingBeamCheck
wellformednesstools.OverlappingGlissandoCheck
wellformednesstools.OverlappingOctavationCheck
wellformednesstools.ShortHairpinCheck
```

- The classes check different aspects of score well-formedness.
- To call these classes use `wellformednesstools.is_well_formed_component()` or `wellformednesstools.tabulate_well_formedness_violations_in_expr()`.

Added new `decoratortools` package.

- This version of the package contains only the `requires` decorator.
- The `requires` decorator will be used in later versions of Abjad to specify the input and output types of functions explicitly.
- This will help in the construction of function- and class-population tools.

Added new `scoretemplatetools` package.

- This version of the package implements the following classes:

```
scoretemplatetools.StringQuartetScoreTemplate
scoretemplatetools.TwoStaffPianoScoreTemplate
```

- Example:

```
>>> from abjad.tools import scoretemplatetools
```

```
>>> template = scoretemplatetools.StringQuartetScoreTemplate()
>>> score = template()
```

```
>>> score
Score-"String Quartet Score"<<1>>
```

```
>>> f(score)
\context Score = "String Quartet Score" <<
    \context StaffGroup = "String Quartet Staff Group" <<
        \context Staff = "First Violin Staff" {
            \clef "treble"
            \context Voice = "First Violin Voice" {
            }
        }
        \context Staff = "Second Violin Voice" {
            \clef "treble"
        }
        \context Staff = "Viola Staff" {
            \clef "alto"
        }
        \context Staff = "Cello Staff" {
            \clef "bass"
        }
    >>
>>
```

- Class usage follows a two-step initialize-then-call pattern.

Added new `rhythmtreetools` package for parsing IRCAM-like RTM syntax.

- This version of the package implements the following function:

```
rhythmtreetools.parse_rtm_syntax.parse_rtm_syntax()
```

- Example:

```
>>> from abjad.tools.rhythmtreetools import parse_rtm_syntax
```

```
>>> rtm = '(1 (1 (1 (1 1)) 1))'
>>> result = parse_rtm_syntax(rtm)
>>> result
FixedDurationTuplet(1/4, [c'8, c'16, c'16, c'8])
```

- Use the `rhythmtreetools` package to turn nested lists of numbers into Abjad tuplets.

Added new `rhythmmakertools` package.

- This version of the package contains the following concrete classes:

```
rhythmmakertools.NoteRhythmMaker
rhythmmakertools.OutputBurnishedTaleaRhythmMaker
rhythmmakertools.OutputIncisedNoteRhythmMaker
rhythmmakertools.OutputIncisedRestRhythmMaker
rhythmmakertools.RestRhythmMaker
rhythmmakertools.TaleaRhythmMaker
rhythmmakertools.DivisionBurnishedTaleaRhythmMaker
rhythmmakertools.DivisionIncisedNoteRhythmMaker
rhythmmakertools.DivisionIncisedRestRhythmMaker
```

- The `rhythmmakertools` package implements a family of related rhythm-making classes.
- Class usage follows a two-step initialize-then-call pattern.

Added new classes to `instrumenttools`.

- Added human voice classes:

```
instrumenttools.BaritoneVoice
instrumenttools.BassVoice
instrumenttools.ContraltoVoice
instrumenttools.MezzoSopranoVoice
instrumenttools.SopranoVoice
instrumenttools.TenorVoice
```

Added new time-interval tree functionality:

- Extended `TimeIntervalTree` with the following public methods:

```
scale_by_rational()
scale_to_rational()
shift_by_rational()
shift_to_rational()
split_at_rationals()
```

- These methods allow time-interval trees to behave more similary to time-intervals.

All score components are now public.

- The following classes are now publically available for the first time:

```
componenttools.Component
contexttools.Context
leaftools.Leaf
```

Further new functionality:

- Added the `marktools.BendAfter` class to model LilyPond's `\bendAfter command`:

```
>>> n = Note(0, 1)
>>> marktools.BendAfter(8)(n)
BendAfter(8.0)(c'1)
>>> f(n)
c'1 - \bendAfter #'8.0
```

- Added public `pair` property to `contexttools.TimeSignatureMark`:

```
>>> time_signature = contexttools.TimeSignatureMark((3, 16))
>>> time_signature.pair
(3, 16)
```

- Added `is_hairpin_token()` to `spannertools.HairpinSpanner` class.

  Hairpin tokens are triples of the form `(x, y, z)` with dynamic tokens `x`, `y` and hairpin shape string `z`. For example `('p', '<', 'f')`.

- Added `resttools.replace_leaves_in_expr_with_rests()`.
- Added `leaftools.replace_leaves_in_expr_with_parallel_voices()`.
- Added `leaftools.replace_leaves_in_expr_with_named_parallel_voices()`.

  Use the functions listed above to replace leaves in an expression with parallel voices containing copies of those leaves in both voices. This is useful for generating stemmed-glissandi structures.

- Added `contexttools.list_clef_names()`:

```
>>> contexttools.list_clef_names()
['alto', 'baritone', 'bass', 'mezzosoprano', 'percussion', 'soprano', 'treble']
```

- Added `find-slots-implementation-inconsistencies` development script.

Changes to end-user functionality:

- Changed `intervaltreetools` to `timeintervaltools`.
- Changed `contexttools.Context.context` to `contexttools.Context.context_name`.
- Calling `bool(Container())` on empty containers now returns false. The previous behavior of the system was to return true. The new behavior better conforms to the Python iterable interface.

- Moved `abjad/docs/scr/make-abjad-api` to `abjad/scr/make-abjad-api`.

3.2.4 Abjad 2.7

Released 2012-02-27. Built from r5100. Implements 221 public classes and 1029 functions totalling 168,000 lines of code.

- Added `lilypondparsertools.LilyPondParser` class, which arses a subset of LilyPond input syntax:

```
>>> from abjad.tools.lilypondparsertools import LilyPondParser
>>> parser = LilyPondParser( )
>>> input = r"\new Staff { c'4 ( d'8 e' fs'2) \fermata }"
>>> result = parser(input)
>>> f(result)
\new Staff {
    c'4 (
    d'8
    e'8
    fs'2 -\fermata )
}
```

  LilyPondParser defaults to English note names, but any of the other languages supported by LilyPond may be used:

```
>>> parser = LilyPondParser('nederlands')
>>> input = '{ c des e fis }'
>>> result = parser(input)
>>> f(result)
{
    c4
    df4
    e4
    fs4
}
```

  Briefly, LilyPondParser understands theses aspects of LilyPond syntax:

  - Notes, chords, rests, skips and multi-measure rests
  - Durations, dots, and multipliers
  - All pitchnames, and octave ticks
  - Simple markup (i.e. `c'4 ^ "hello!"`)
  - Most articulations
  - Most spanners, including beams, slurs, phrasing slurs, ties, and glissandi
  - Most context types via `\new` and `\context`, as well as context ids (i.e. `\new Staff = "foo" { }`)
  - Variable assignment (i.e. `global = { \time 3/4 } \new Staff { \global }`)
  - Many music functions: - `\acciaccatura` - `\appoggiatura` - `\bar` - `\breathe` - `\clef` - `\grace` - `\key` - `\transpose` - `\language` - `\makeClusters` - `\mark` - `\oneVoice` - `\relative` - `\skip` - `\slashedGrace` - `\time` - `\times` - `\transpose` - `\voiceOne`, `\voiceTwo`, `\voiceThree`, `\voiceFour`

  LilyPondParser currently **DOES NOT** understand many other aspects of LilyPond syntax:

  - `\markup`
  - `\book`, `\bookpart`, `\header`, `\layout`, `\midi` and `\paper`
  - `\repeat` and `\alternative`
  - Lyrics
  - `\chordmode`, `\drummode` or `\figuremode`

  - Property operations, such as `\override`, `\revert`, `\set`, `\unset`, and `\once`
  - Music functions which generate or extensively mutate musical structures
  - Embedded Scheme statements (anything beginning with #)

- Added `iotools.p( )`, for conveniently parsing LilyPond syntax:

```
>>> result = p(r"\new Staff { c'4 d e f }")
>>> f(result)
\new Staff {
    c'4
    d4
    e4
    f4
}
```

- Added `schemetools.Scheme`, as a more robust replacement for nearly all other `schemetools` classes:

```
>>> from abjad.tools.schemetools import Scheme
>>> print Scheme(True).format
##t
>>> print Scheme('a', 'list', 'of', 'strings').format
#(a list of strings)
>>> print Scheme(('simulate', 'a', 'vector'), quoting="'#").format
#'#(simulate a vector)
>>> print Scheme('a', ('nested', ('data', 'structure'))).format
#(a (nested (data structure))
```

- Removed deprecated `schemetools` classes:

  - `SchemeBoolean`
  - `SchemeFunction`
  - `SchemeNumber`
  - `SchemeString`
  - `SchemeVariable`

  In all cases, simply use `schemetools.Scheme` instead.

- Reimplemented MarkupCommand.

  The new implementation is initialized from a command-name, and a variable-size list of arguments. Arguments which are lists or tuples will be enclosed in curly-braces:

```
>>> from abjad.tools.markuptools import MarkupCommand
>>> bold = MarkupCommand('bold', ['two', 'words'])
>>> rotate = MarkupCommand('rotate', 60, bold)
>>> triangle = MarkupCommand('triangle', False)
>>> concat = MarkupCommand('concat', ['one word', rotate, triangle])
>>> print concat.format
\concat { #"one word" \rotate #60 \bold { two words } \triangle ##f }
```

- Added `contexttools.TempoMarkInventory`, which models an ordered list of tempo marks:

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

  Inherits from list. Allows initialization, append and extent on tempo mark tokens.

- Added new `pitchtools.PitchRangeInventory` class.

  The class acts as an ordered list of PitchRange objects.

  The purpose of the class is to model something like palettes of different pitches available in all part of a score:

```
>>> pitchtools.PitchRangeInventory(['[C3, C6]', '[C4, C6]'])
PitchRangeInventory([PitchRange('[C3, C6]'), PitchRange('[C4, C6]')])
```

The class inherits from list.

- Added `sequencetools.all_are_pairs()` predicate:

```
>>> from abjad.tools.sequencetools import all_are_pairs
>>> all_are_pairs([(1, 2), (3, 4), (5, 6)])
True
```

- Added `sequencetools.all_are_pairs_of_types()` predicate:

```
>>> from abjad.tools.sequencetools import all_are_pairs_of_types
>>> all_are_pairs_of_types([('a', 1.4), ('b', 2.3), ('c', 1.5)], str, float)
True
```

- Added `stringtools.is_underscore_delimited_lowercase_file_name_with_extension()` string predicate:

```
>>> stringtools.is_underscore_delimited_lowercase_file_name_with_extension('foo_bar.blah')
True
```

- Added `iotools.is_underscore_delimited_file_name()` string predicate.

  Returns true on any underscore-delimited lowercase string.

  Also returns trun on an underscore-delimtied lowercase string terminated with an extension.

```
>>> stringtools.is_underscore_delimited_lowercase_file_name('foo_bar.py')
True

>>> stringtools.is_underscore_delimited_lowercase_file_name('foo_bar')
True
```

- Added `ImpreciseTempoError` to exceptions.
- Added `LilyPondParserError` to exceptions.
- Added `scr/devel/fix-test-cases`. The script is a two-line wrapper around the following other two scripts:
  - `scr/devel/fix-test-case-names`
  - `scr/devel/fix-test-case-numbers`
- Extended `Container` to use `LilyPondParser` to parse input strings.
- Extended `contexttools.InstrumentMark`, `scoretools.Performer` and `markuptools.Markup` with `__hash__` equality.

  Now, if two instances compare equally (via ==), their hashes also compare equally, allowing for more intuitive use of these classes as dictionary keys.
- Extended `contexttools.TempoMark` with textual indications and tempo ranges You may instantiate as normal, or in some new combinations:

```
>>> from abjad.tools.contexttools import TempoMark
>>> t = TempoMark('Langsam', Duration(1, 4), (52, 57))
>>> t = TempoMark('Langsam')
>>> t = TempoMark((1, 4), (52, 57))
```

  In addition to its new read/write "textual_indication" attribute, TempoMark now also exposes a read-only "is_imprecise" property, which returns True if the mark cannot be expressed simply as duration=units_per_minute. Arithmetic operations on TempoMarks will now raise ImpreciseTempoErrors if any mark involved is imprecise.
- Extended tempo marks to be able to initialize from 'tempo mark tokens'. A tempo mark token is a length-2 or length-3 tuple of tempo mark arguments.
- Extended tempo mark with `is_tempo_mark_token()` method:

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

- Extended case-testing `iotools` string predicates to allow digits.

  Functions changed:

  - `stringtools.is_space_delimited_lowercase_string`
  - `stringtools.is_underscore_delimited_lowercase_file_name`
  - `stringtools.is_lowercamelcase_string`
  - `stringtools.is_uppercamelcase_string`
  - `stringtools.is_underscore_delimited_lowercase_string`
  - `stringtools.is_underscore_delimited_lowercase_file_name_with_extension`

- Extended `lilypondfiletools.NonattributedBlock` with `is_formatted_when_empty` read-write property. `lilypondfiletools.ScoreBlock` no longer formats when empty, by default.
- Extended `marktools.BarLine` with `format_slot` keyword.
- Extended `pitchtools.PitchRange` class with read-only `pitch_range_name` and `pitch_range_name_markup` attributes.
- Extended `scoretools.InstrumentationSpecifier` with read-only `performer_name_string` attribute.
- Extended all `beamtools.Beam-`, `Slur-` and `Hairpin-``related spanner classes, as well as ``tietools.TieSpanner` with an optional ``direction` keyword:

```
>>> c = Container("c'4 d'4 e'4 f'4")
>>> spanner = spannertools.SlurSpanner(c[:], 'up')
>>> f(c)
{
    c'4 ^ (
    d'4
    e'4
    f'4 )
}
```

  The direction options are exactly the same as for `Articulation` and `Markup`: `'up'`, `'^'`, `'down'`, `'_'`, `'neutral'`, `'-'` and `None`.

- Extended `tonalitytools.Scale` with `create_named_chromatic_pitch_set_in_pitch_range()` method.
- Changed `tuplettools.FixedDurationTuplet.multiplier` to return fraction instead of duration.
- Renamed attributes, methods and functions throughout `intervaltreetools`:
  - `centroid` => `center` (except where a weighted mean is actually used)
  - `high` => stop``
  - `high_min` => earliest_stop``
  - `high_max` => latest_stop``
  - `low` => `start`
  - `low_min` => `earliest_start`
  - `low_max` => `latest_start`
  - `magnitude` => `duration`

  This both clarifies the API, and prevents shadowing of Python's builtin `min()` and `max()`.

- Renamed `marktools.Articulation.direction_string` => `marktools.Articulation.direction`.
- Renamed `markuptools.Markup.direction_string` => `markuptools.Markup.direction`.

- Renamed `tuplettools.Tuplet.ratio` to `tuplettools.Tuplet.ratio_string`.
- Renamed `scr/devel/find-nonalphabetized-method-names` to `scr/devel/find-nonalphabetized-class-attributes`.
- Improved `scr/devel/find-nonalphabetzied-methods`.
- Updated literature examples to match API changes.
- Removed ancient `stafftools.make_invisible_staff()`.
- Added `text_editor` key to user config dictionary (in `~/.abjad/config.py`).
- Improved `__repr__` strings of `tonalitytools.Mode` and `tonalitytools.Scale`.
- `contexttools.TempoMark` `__repr__` now shows `__repr__` version of duration instead of string version of duration.
- `scr/devel/abj-grp` no longer excludes lines of code that include the string `'svn'`.

3.2.5 Abjad 2.6

Released 2012-01-29. Built from r4979. Implements 197 public classes and 941 public functions totalling 153,000 lines of code.

- Added top-level `decorators` directory with `requires` decorator. The `requires` decorator renders the following two function definitions equivalent:

```
from abjad.tools.decoratortools import requires
```

```
@requires(int)
def foo(x):
    return x ** 2
```

```
def foo(x):
    assert isinstance(x, int)
    return x ** 2
```

- Added new classes to `scoretools`:

```
scoretools.InstrumentationSpecifier
scoretools.Performer
```

- Added `scoretools.list_performer_names()`:

```
>>> for name in scoretools.list_performer_names()[:10]:
...     name
...
'accordionist'
'bassist'
'bassoonist'
'cellist'
'clarinetist'
'flutist'
'guitarist'
'harpist'
'harpsichordist'
'hornist'
```

- Added `scoretools.list_primary_performer_names()`.
- Added `measuretools.measure_to_one_line_input_string()`:

```
>>> measure = Measure((3, 4), "c4 d4 e4")
```

```
>>> measure
Measure(3/4, [c4, d4, e4])
```

```
>>> measuretools.measure_to_one_line_input_string(measure)
"Measure((3, 4), 'c4 d4 e4')"
```

- Added new classes to `instrumenttools`:

```
SopraninoSaxophone
SopranoSaxophone
AltoSaxophone
TenorSaxophone
BaritoneSaxophone
BassSaxophone
ContrabassSaxophone
```

```
ClarinetInA
```

```
AltoTrombone
BassTrombone
```

```
Harpsichord
```

- Added known untuned percussion:

```
>>> for name in instrumenttools.UntunedPercussion.known_untuned_percussion[:10]:
...     print name
...
agogô
anvil
bass drum
bongo drums
cabasa
cajón
castanets
caxixi
claves
conga drums
```

- Added `_Instrument.get_default_performer_name()`:

```
>>> bassoon = instrumenttools.Bassoon()
```

```
>>> bassoon.get_default_performer_name()
'bassoonist'
```

- Added `_Instrument.get_performer_names()`:

```
>>> bassoon.get_performer_names()
['instrumentalist', 'reed player', 'double reed player', 'bassoonist']
```

- Added read / write `_Instrument.pitch_range`:

```
>>> marimba.pitch_range = (-24, 36)
>>> marimba.pitch_range
PitchRange('[C2, C7]')
```

- Added read-only `_Instrument.traditional_pitch_range`:

```
>>> marimba = instrumenttools = instrumenttools.Marimba()
>>> marimba.traditional_pitch_range
PitchRange('[F2, C7]')
```

- Added `instrumenttools.list_instruments()`:

```
>>> for instrument_name in instrumenttools.list_instrument_names()[:10]:
...     instrument_name
...
'accordion'
'alto flute'
'alto saxophone'
'alto trombone'
'clarinet in B-flat'
```

```
'baritone saxophone'
'bass clarinet'
'bass flute'
'bass saxophone'
'bass trombone'
```

- Added other functions to `instrumenttools`:

```
instrumenttools.list_primary_instrument_names()
instrumenttools.list_secondary_instrument_names()
```

- Added new class to `lilypondfiletools`:

```
ContextBlock
```

- Added `pitchtools.is_symbolic_pitch_range_string()`:

```
>>> pitchtools.is_symbolic_pitch_range_string('[A0, C8]')
True
```

- Added `pitchtools.pitch_class_octave_number_string_to_chromatic_pitch_name()`:

```
>>> pitchtools.pitch_class_octave_number_string_to_chromatic_pitch_name('A#4')
"as'"
```

- Added `pitchtools.symbolic_accidental_string_to_alphabetic_accidental_string_abbreviat`

```
>>> pitchtools.alphabetic_accidental_abbreviation_to_symbolic_accidental_string('tqs')
'#+'
```

- Added other new functions to `pitchtools`:

```
pitchtools.alphabetic_accidental_abbreviation_to_symbolic_accidental_string()
pitchtools.is_smbolic_accidental_string()
pitchtools.is_pitch_class_octave_number_string()
```

- Added `stringtools.string_to_strict_directory_name()`:

```
>>> stringtools.string_to_strict_directory_name('Déja vu')
'deja_vu'
```

- Added `stringtools.strip_diacritics_from_binary_string()`:

```
>>> binary_string = 'Dvořák'
>>> stringtools.strip_diacritics_from_binary_string(binary_string)
'Dvorak'
```

- Added other new functions to `iotools`:

```
stringtools.capitalize_string_start()
iotools.is_space_delimited_lowercamelcase_string()
iotools.is_underscore_delimited_lowercamelcase_package_name()
iotools.is_underscore_delimited_lowercamelcase_string()
stringtools.is_lowercamelcase_string()
stringtools.is_uppercamelcase_string()
stringtools.space_delimited_lowercase_to_uppercamelcase()
stringtools.uppercamelcase_to_space_delimited_lowercase()
stringtools.uppercamelcase_to_underscore_delimited_lowercase()
```

- Added new functions to `mathtools`:

```
mathtools.is_positive_integer_power_of_two()
mathtools.is_integer_equivalent_expr()
```

- Added sequence type-checking predicates:

```
chordtools.all_are_chords()
containertools.all_are_containers()
durationtools.all_are_duration_tokens()
durationtools.all_are_durations()
gracetools.all_are_grace_containers()
```

```
leaftools.all_are_leaves()
markuptools.all_are_markup()
measuretools.all_are_measures()
notetools.all_are_notes()
pitcharraytools.all_are_pitch_arrays()
pitchtools.all_are_named_chromatic_pitch_tokens()
resttools.all_are_rests()
scoretools.all_are_scores()
sievetools.all_are_residue_class_expressions()
skiptools.all_are_skips()
spannertools.all_are_spanners()
stafftools.all_are_staves()
tuplettools.all_are_tuplets()
```

- Extended `NamedChromaticPitch` to allow initialization from pitch-class / octave number strings:

```
>>> pitchtools.NamedChromaticPitch('C#2')
NamedChromaticPitch('cs,')
```

- Extended `PitchRange` to allow initialization from symbolic pitch range strings:

```
>>> pitchtools.PitchRange('[A0, C8]')
PitchRange('[A0, C8]')
```

- Extended `PitchRange` to allow initialization from pitch-class / octave number strings:

```
>>> pitchtools.PitchRange('A0', 'C8')
PitchRange('[A0, C8]')
```

- Extended `leaftools.is_bar_line_crossing_leaf()` to work when no explicit time signature mark is found.
- Extended `Markup` to be able to function as a top-level `LilyPondFile` element.
- Extended instruments with `is_primary` and `is_secondary` attributes.
- Extended instruments with `instrument_name` and `instrument_name_markup` attributes.
- Extended instruments with `short_instrument_name` and `short_instrument_name_markup` attributes.
- Extended `iotools.write_expr_to_ly()` and `iotools.write_expr_to_pdf()` with `'tagline'` keyword.
- Extended `replace-in-files` script to skip `.text`, `.ly` and `.txt` files.
- Renamed `Accidental.symbolic_string` to `Accidental.symbolic_accidental_string`.
- Renamed `Accidental.alphabetic_string` to `Accidental.alphabetic_accidental_abbreviation`.
- Fixed bug in `iotools.play()`.
- Fixed bug in `quantizationtools` regarding quantizing a stream of `QEvents` directly.

3.2.6 Abjad 2.5

Released 2011-09-22. Built from r4803.

- Added `get_leaf_in_expr_with_minimum_duration()` function to `leaftools`.
- Added `get_leaf_in_expr_with_maximum_duration()` function to `leaftools`.
- Added `are_relatively_prime()` function to `mathtools`.
- Added `CyclicTree` class to `sequencetools`.
- Added `get_next_n_nodes_at_leve(n, level)` method to `sequencetools.Tree`.
- Extended spanners to sort by repr.
- Renamed `lilyfiletools` to `lilypondfiletools`.

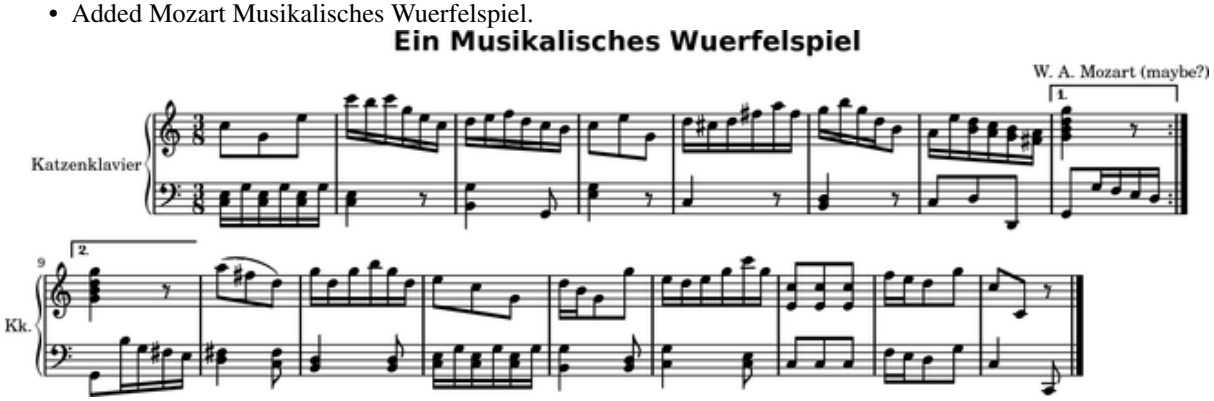
- Renamed `lilyfiletools.LilyFile` to `lilypondfiletools.LilyPondFile`.
- Renamed `lilyfiletools.make_basic_lily_file()` to `lilypondfiletools.make_basic_lilypond_f`

Note that the three renames change user syntax. Composers working with the `lilypondfiletools` module should update their score code.

3.2.7 Abjad 2.4

Released 2011-09-12. Built from r4769.

- Added Mozart Musikalisches Wuerfelspiel.



- Added new `Tree` class to `sequencetools` to work with sequences whose elements have been grouped into arbitrarily many levels of containment.
- Added new `BarLine` class to `marktools` package.
- Added new `HorizontalBracketSpanner` to `spannertools` package.
- Improved `schemetools.SchemePair` handling.
- Extended `LilyPondFile` blocks with double underscore-delimited attributes.

3.2.8 Abjad 2.3

Released 2011-09-04. Built from r4747.

Filled out the API for working with marks:

```
marktools.attach_articulations_to_components_in_expr()
marktools.detach_articulations_attached_to_component()
marktools.get_articulations_attached_to_component()
marktools.get_articulation_attached_to_component()
marktools.is_component_with_articulation_attached()
```

These five type of functions are now implemented for the following marks:

```
marktools.Annotation
marktools.Articulation
marktools.LilyPondCommandMark
marktools.LilyPondComment
marktools.StemTremolo
```

The same type of functions are likewise implemented for the following context marks:

```
contexttools.ClefMark
contexttools.DynamicMark
contexttools.InstrumentMark
contexttools.KeySignatureMark
contexttools.StaffChangeMark
contexttools.TempoMark
contexttools.TimeSignatureMark
```

- Extended `Container.extend()` to allow for LilyPond input strings. You can now say `container.extend("c'4 d'4 e'4 f'4")`.
- Added public `parent` attribute to all components. You can now say `note.parent`. The attribute is read-only.
- Added `cfgtools.list_package_dependency_version()`.
- Added `py.test` and `Sphinx` dependencies to the Abjad package.
- Added LilyPond command mark chapter to reference manual
- Renamed `cfgtools` to `configurationtools`.
- Renamed `durtools` to `durationtools`.
- Renamed `metertools` to `timesignaturetools`.
- Renamed `seqtools` to `sequencetools`.
- Renamed `Mark.attach_mark()` to `Mark.attach()`.
- Renamed `Mark.detach_mark()` to `Mark.detach()`.
- Renamed `marktools.Comment` to `marktools.LilyPondComment`. This matches `marktools.LilyPondCommandMark`.
- Removed `contexttools.TimeSignatureMark(3, 8)` initialization. You must now say `contexttools.TimeSignatureMark((3, 8))` instead. This parallels the initialization syntax for rests, skips and measures.

3.2.9 Abjad 2.2

Released 2011-08-30. Built from r4677.

- Added articulations chapter to reference manual.
- Reordered the way in which Abjad determines the value of the `HOME` environment variable.
- Updated `scr/devel/replace-in-files` to avoid image files.
- Updated `iotools.log()` to call operating-specific text editor.

3.2.10 Abjad 2.1

Released 2011-08-21. Built from r4655.

- Updated instrument mark `repr` to display target context when instrument mark is attached.
- Extended `scr/abj` and `scr/abjad` to display Abjad version and revision numbers on startup.

3.2.11 Abjad 2.0

Released 2011-08-17. Built from r4638.

Abjad 2.0 is the first public release of Abjad in more than two years. The new release of the system more than doubles the number of classes, functions and packages available in Abjad.

- The API has been cleaned up and completely reorganized. Features have been organized into a collection of 39 different libraries:

```
cfgtools/          instrumenttools/   mathtools/         resttools/         tempotools/
chordtools/        intervaltreetools/ measuretools/      schemetools/       threadtools/
componenttools/    iotools/           metertools/        scoretools/        tietools/
containertools/    layouttools/       musicxmltools/     seqtools/          tonalitytools/
contexttools/      leaftools/         notetools/         sievetools/        tuplettools/
```

```
durtools/          lilyfiletools/     pitcharraytools/   skiptools/         verticalitytools/
gracetools/        marktools/         pitchtools/        spannertools/      voicetools/
importtools/       markuptools/       quantizationtools/ stafftools/
```

- The name of almost every function in the public API has been changed to better indication what the function does. While this has the effect of making Abjad 2.0 largely non-backwards compatible with code written in Abjad 1.x, the longer and much more explicit function names in Abjad 2.0 make code used to structure complex scores dramatically easier to maintain and understand.
- The `contexttools`, `instrumenttools`, `intervaltreetools`, `lilyfiletools`, `marktools`, `pitcharraytools`, `quantizationtools`, `sievetools`, `tonalitytools` and `verticalitytools` packages are completely new.
- The classes implemented in the `contexttools` and `marktools` packages provide an object-oriented interfaces to clefs, time signatures, key signatures, articulations, tempo marks and other symbols stuck to the outside of the hierarchical score tree. The classes implemented in `contexttools` and `marktools` model information outside the score tree much the way that the classes implemented in `spannertools` implement object-oriented interfaces to beams, brackets, hairpins, glissandi and other line-like symbols.
- The `instrumenttools` package provides an object-oriented model of most of the conventional instruments of the orchestra.
- The `intervaltreetools` package implements a custom way of working with chunks of score during composition.
- The `lilyfiletools` package implements an object-oriented interface to arbitrarily structured LilyPond input files.

Part II

Examples

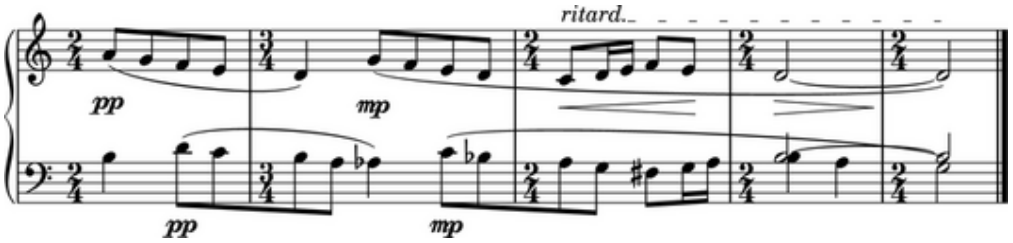
CHAPTER

FOUR

BARTÓK: *MIKROKOSMOS*

This example reconstructs the last five measures of Bartók's "Wandering" from *Mikrokosmos*, volume III. The end result is just a few measures long but covers the basic features you'll use most often in Abjad.

Here is what we want to end up with:



4.1 The score

We'll construct the fragment top-down from containers to notes. We could have done it the other way around but it will be easier to keep the big picture in mind this way. Later, you can rebuild the example bottom-up as an exercise.

First let's create an empty score with a pair of staves connected by a brace:

```
>>> score = Score([])
>>> piano_staff = scoretools.PianoStaff([])
>>> upper_staff = Staff([])
>>> lower_staff = Staff([])
```

```
>>> piano_staff.append(upper_staff)
>>> piano_staff.append(lower_staff)
>>> score.append(piano_staff)
```

Here we create an empty score and assign it to the `score` variable. Then we create an empty piano staff assigned to the `piano_staff` variable and two empty staves assigned to the `upper_staff` and `lower_staff` variables. Finally, we append the two staves to the piano staff and the piano staff to the score.

4.2 The measures

Now let's add some measures to our score:

```
>>> m1 = Measure((2, 4), [])
>>> m2 = Measure((3, 4), [])
>>> m3 = Measure((2, 4), [])
>>> m4 = Measure((2, 4), [])
>>> m5 = Measure((2, 4), [])
```

```
>>> upper_measures = [m1, m2, m3, m4, m5]
>>> lower_measures = componenttools.copy_components_and_covered_spanners(upper_measures)
```

```
>>> upper_staff.extend(upper_measures)
>>> lower_staff.extend(lower_measures)
```

The lower measures are copies of the upper measures.

Note that we add lists of measures to staves with `extend()`. This is because `extend()` is used for adding many objects to an iterable at once while `append()` is used to add only one object at a time.

4.3 The notes

Now let's add some notes. We begin with the upper staff:

```
>>> upper_measures[0].extend([Note(i, (1, 8)) for i in [9, 7, 5, 4]])
```

```
>>> upper_measures[1].extend(notetools.make_notes([2, 7, 5, 4, 2], [(1, 4)] + [(1, 8)] * 4))
```

```
>>> notes = notetools.make_notes([0, 2, 4, 5, 4], [(1, 8), (1, 16), (1, 16), (1, 8), (1, 8)])
>>> upper_measures[2].extend(notes)
```

```
>>> upper_measures[3].append(Note("d'2"))
```

```
>>> upper_measures[4].append(Note("d'2"))
```

Now let's add notes to the lower staff. This will be a more intricate process than that needed for the upper staff. We added notes directly to the measures of the upper staff. But this will not be possible for the lower staff because of the simultaneous voices the lower staff contains.

We add notes to the lower staff measure by measure:

```
>>> main_voice_m1 = Voice("b4 d'8 c'8")
>>> main_voice_m1.name = 'main_voice'
>>> lower_measures[0].append(main_voice_m1)
```

```
>>> main_voice_m2 = Voice("b8 a8 af4 c'8 bf8")
>>> main_voice_m2.name = 'main_voice'
>>> lower_measures[1].append(main_voice_m2)
```

```
>>> main_voice_m3 = Voice("a8 g8 fs8 g16 a16")
>>> main_voice_m3.name = 'main_voice'
>>> lower_measures[2].append(main_voice_m3)
```

Notice that we give the same name to the three voices contained in the first three measures of the lower staff.

It is in the last two measures of the lower staff where Bartók writes two voices at once. We'll name the second of these two voices the *appendix_voice*:

```
>>> appendix_voice_m4 = Voice([Note("b2")])
>>> appendix_voice_m4.name = 'appendix_voice'
>>> lilypond_command_mark = marktools.LilyPondCommandMark('voiceOne')
>>> lilypond_command_mark.attach(appendix_voice_m4)
LilyPondCommandMark('voiceOne')(Voice-"appendix_voice"{1})
```

```
>>> main_voice_m4 = Voice("b4 a4")
>>> main_voice_m4.name = 'main_voice'
>>> lilypond_command_mark = marktools.LilyPondCommandMark('voiceTwo')
>>> lilypond_command_mark.attach(main_voice_m4)
LilyPondCommandMark('voiceTwo')(Voice-"main_voice"{2})
```

```
>>> container = Container([appendix_voice_m4, main_voice_m4])
>>> container.is_parallel = True
>>> lower_measures[3].append(container)
```

The LilyPond `\voiceOne` and `\voiceTwo` commands determine the direction of the stems in different voices.

Note that we must put both voices in a parallel container because they occur at the same time in the score. We do this by creating an Abjad container and then setting the `is_parallel` attribute of the container to true.

We now do a similar thing for the last measure:

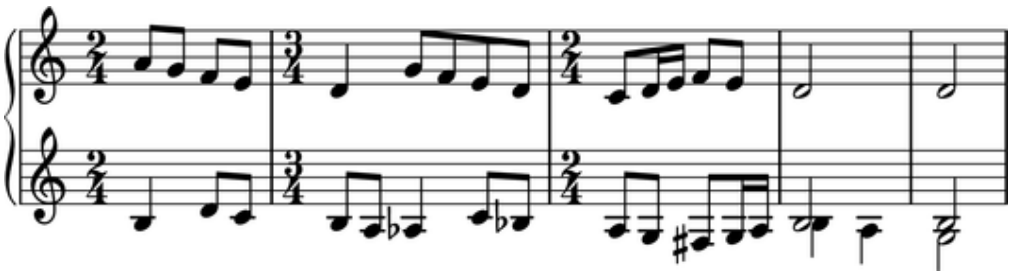
```
>>> appendix_voice_m5 = Voice("b2")
>>> appendix_voice_m5.name = 'appendix_voice'
>>> lilypond_command_mark = marktools.LilyPondCommandMark('voiceOne')
>>> lilypond_command_mark.attach(appendix_voice_m5)
LilyPondCommandMark('voiceOne')(Voice-"appendix_voice"{1})
```

```
>>> main_voice_m5 =  Voice("g2")
>>> main_voice_m5.name = 'main_voice'
>>> lilypond_command_mark = marktools.LilyPondCommandMark('voiceTwo')
>>> lilypond_command_mark.attach(main_voice_m5)
LilyPondCommandMark('voiceTwo')(Voice-"main_voice"{1})
```

```
>>> container = Container([appendix_voice_m5, main_voice_m5])
>>> container.is_parallel = True
>>> lower_measures[4].append(container)
```

Here's our work so far:

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



4.4 The details

Ok, let's add the details. First, notice that the bottom staff has a treble clef just like the top staff. Let's change that:

```
>>> contexttools.ClefMark('bass')(lower_staff)
ClefMark('bass')(Staff{5})
```

Now let's add dynamic marks. For the top staff, we'll add them to the first note of the first measure and the second note of the second measure. For the bottom staff, we'll add dynamic markings to the second note of the first measure and the fourth note of the second measure.

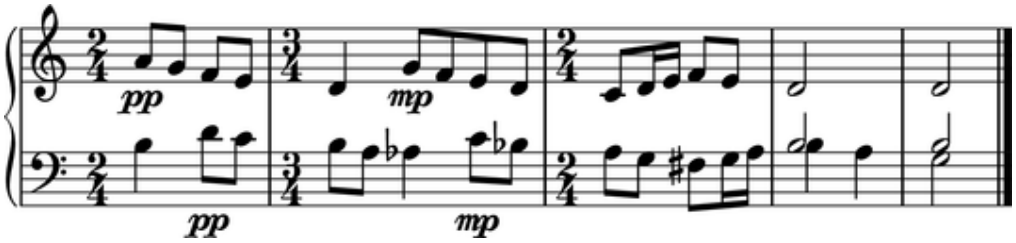
```
>>> contexttools.DynamicMark('pp')(upper_measures[0][0])
DynamicMark('pp')(a'8)
>>> contexttools.DynamicMark('mp')(upper_measures[1][1])
DynamicMark('mp')(g'8)
>>> contexttools.DynamicMark('pp')(lower_measures[0][0][1])
DynamicMark('pp')(d'8)
>>> contexttools.DynamicMark('mp')(lower_measures[1][0][3])
DynamicMark('mp')(c'8)
```

Let's add a double bar to the end of the piece:

```
>>> bar_line = marktools.BarLine('|.')
>>> bar_line.attach(lower_staff.leaves[-1])
BarLine('|.')(g2)
```

And see how things are coming out:

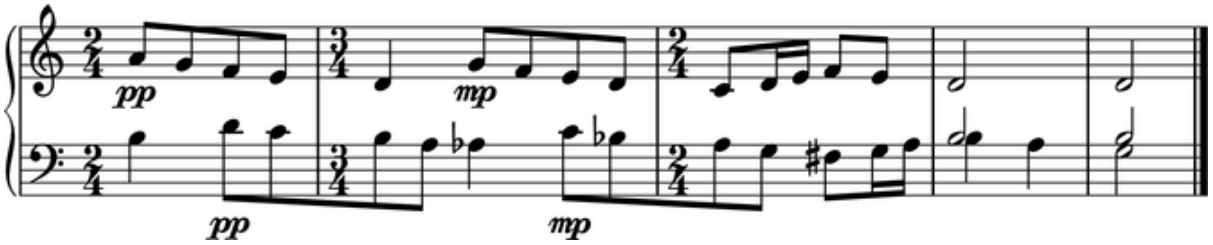
```
>>> show(score)
```



Notice that the beams of the eighth and sixteenth notes appear as you would usually expect: grouped by beat. We get this for free thanks to LilyPond's default beaming algorithm. But this is not the way Bartók notated the beams. Let's set the beams as Bartók did with some crossing the bar lines:

```
>>> beamtools.BeamSpanner(upper_measures[0])
BeamSpanner(|2/4(4)|)
>>> beamtools.BeamSpanner(lower_staff.leaves[1:5])
BeamSpanner(d'8, c'8, b8, a8)
>>> beamtools.BeamSpanner(lower_staff.leaves[6:10])
BeamSpanner(c'8, bf8, a8, g8)
```

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



Now some slurs:

```
>>> spannertools.SlurSpanner(upper_staff.leaves[0:5])
SlurSpanner(a'8, g'8, f'8, e'8, d'4)
>>> spannertools.SlurSpanner(upper_staff.leaves[5:])
SlurSpanner(g'8, f'8, ... [7] ..., d'2, d'2)
>>> spannertools.SlurSpanner(lower_staff.leaves[1:6])
SlurSpanner(d'8, c'8, b8, a8, af4)
>>> spannertools.SlurSpanner(lower_staff.leaves[6:13] + (main_voice_m4, main_voice_m5))
SlurSpanner(c'8, bf8, ... [5] ..., {b4, a4}, {g2})
```

Hairpins:

```
>>> spannertools.CrescendoSpanner(upper_staff.leaves[-7:-2])
CrescendoSpanner(c'8, d'16, e'16, f'8, e'8)
>>> spannertools.DecrescendoSpanner(upper_staff.leaves[-2:])
DecrescendoSpanner(d'2, d'2)
```

A ritardando marking above the last seven notes of the upper staff:

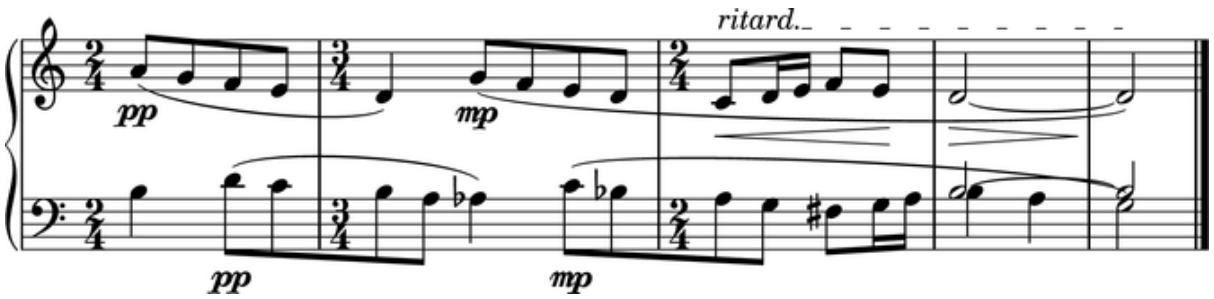
```
>>> text_spanner = spannertools.TextSpanner(upper_staff.leaves[-7:])
>>> text_spanner.override.text_spanner.bound_details__left__text = markuptools.Markup('ritard.')
```

And ties connecting the last two notes in each staff:

```
>>> tietools.TieSpanner(upper_staff[-2:])
TieSpanner(|2/4(1)|, |2/4(1)|)
>>> tietools.TieSpanner([appendix_voice_m4[0], appendix_voice_m5[0]])
TieSpanner(b2, b2)
```

The final result:

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


ritard.
pp
mp
pp
mp

CHAPTER

FIVE

FERNEYHOUGH: *UNSICHTBARE FARBEN*

Note: Explore the *abjad/demos/ferneyhough/* directory for the complete code to this example, or import it into your Python session directly with:

- *from abjad.demos import ferneyhough*

Mikhïal Malt analyzes the rhythmic materials of Ferneyhough's *Unsichtbare Farben* in *The OM Composer's Book 2*.

Malt explains that Ferneyhough used OpenMusic to create an "exhaustive catalogue of rhythmic cells" such that:

1. They are subdivided into two pulses, with proportions from `1/1` to `1/11`.
2. The second pulse is subdivided successively by `1`, `2`, `3`, `4`, `5` and `6`.

Let's recreate Malt's results in Abjad.

5.1 The proportions

First we define proportions:

```
>>> proportions = [(1, n) for n in range(1, 11 + 1)]
```

```
>>> proportions
[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (1, 10), (1, 11)]
```

5.2 The transforms

Then we define a helper function:

```
def divide_tuplet(tuplet, n):
    last_tie_chain = tietools.get_tie_chain(tuplet[-1])
    proportions = n * [1]
    tietools.tie_chain_to_tuplet_with_ratio(last_tie_chain, proportions)
```

5.3 The rhythms

We set the duration of each tuplet equal to a quarter note:

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

And then we make the rhythms:

```
>>> music = []
>>> for proportion in proportions:
...     tuplets = []
...     for n in range(1, 6 + 1):
...         tuplet = tuplettools.make_tuplet_from_duration_and_ratio(duration, proportion)
...         divide_tuplet(tuplet, n)
...         tuplets.append(tuplet)
...     music.extend(tuplets)
...
```

5.4 The score

We make the score:

```
>>> staff = stafftools.RhythmicStaff(music)
>>> time_signature = contexttools.TimeSignatureMark((1, 4))(staff)
>>> score = Score([staff])
```

And then configure it:

```
>>> score.set.proportional_notation_duration = schemetools.SchemeMoment(1, 56)
>>> score.set.tuplet_full_length = True
>>> score.override.bar_line.stencil = False
>>> score.override.bar_number.transparent = True
>>> score.override.spacing_spanner.uniform_stretching = True
>>> score.override.spacing_spanner.strict_note_spacing = True
>>> score.override.time_signature.stencil = False
>>> score.override.tuplet_bracket.padding = 2
>>> score.override.tuplet_bracket.staff_padding = 4
>>> score.override.tuplet_number.text = schemetools.Scheme('tuplet-number::calc-fraction-text')
```

5.5 The LilyPond file

Finally we insert the score into a LilyPond file:

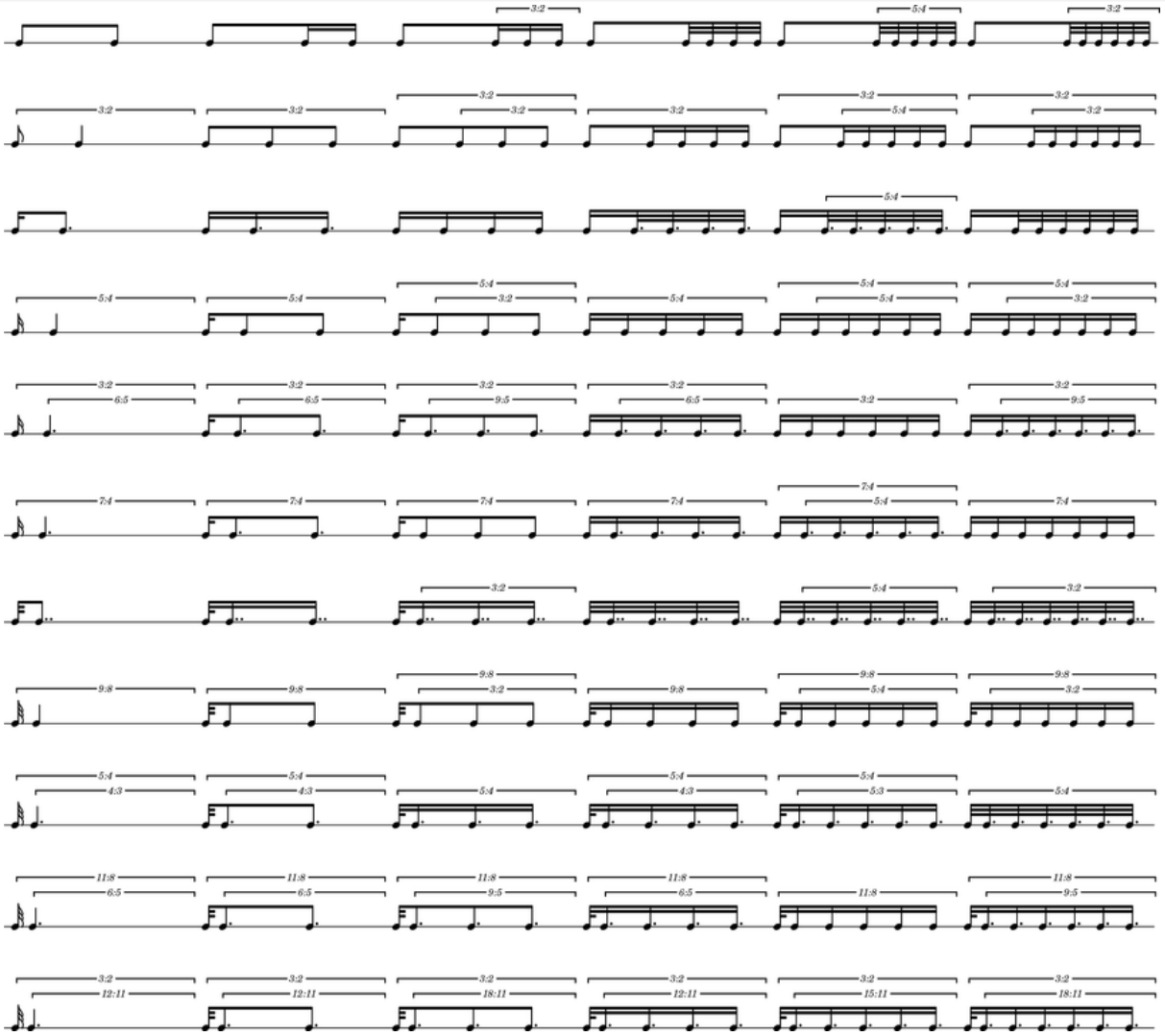
```
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
```

And then configure it:

```
>>> lilypond_file.default_paper_size = '11x17', 'portrait'
>>> lilypond_file.global_staff_size = 12
>>> lilypond_file.layout_block.indent = 0
>>> lilypond_file.layout_block.ragged_right = True
>>> lilypond_file.paper_block.ragged_bottom = True
>>> lilypond_file.paper_block.system_system_spacing = layouttools.make_spacing_vector(0, 0, 8, 0)
```

Which looks like this:

```
>>> show(lilypond_file)
```



CHAPTER

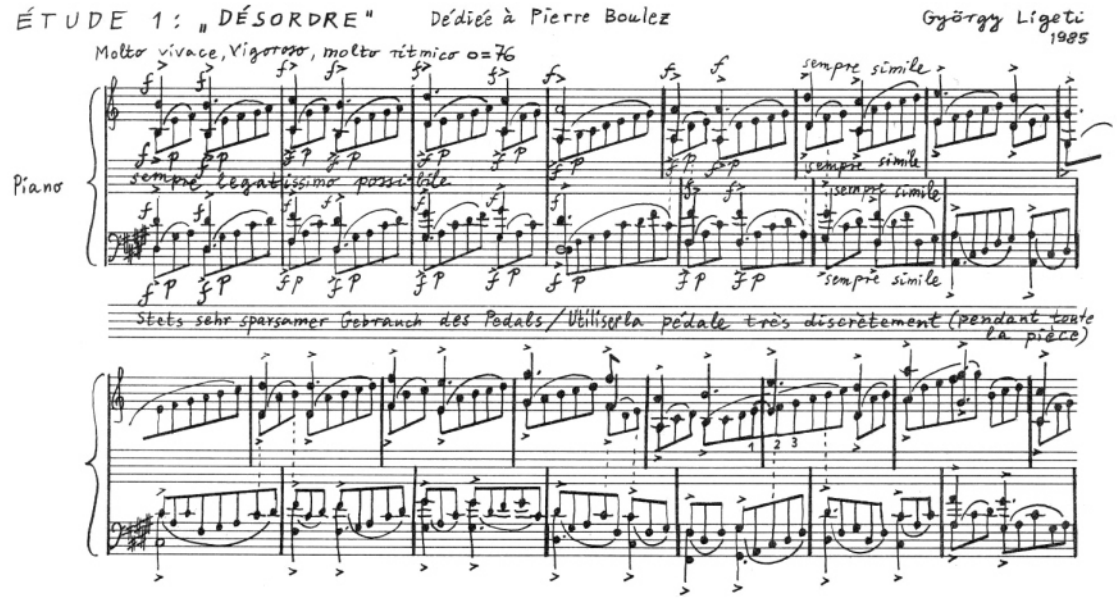
SIX

LIGETI: *DÉSORDRE*

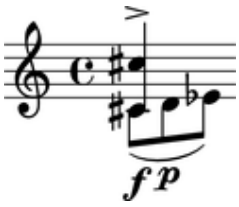
Note: Explore the *abjad/demos/desordre/* directory for the complete code to this example, or import it into your Python session directly with:

- *from abjad.demos import desordre*

This example demonstrates the power of exploiting redundancy to model musical structure. The piece that concerns us here is Ligeti's *Désordre*: the first piano study from Book I. Specifically, we will focus on modeling the first section of the piece:



The redundancy is immediately evident in the repeating pattern found in both staves. The pattern is hierarchical. At the smallest level we have what we will here call a *cell*:



There are two of these cells per measure. Notice that the cells are strictly contained within the measure (i.e., there are no cells crossing a bar line). So, the next level in the hierarchy is the measure. Notice that the measure sizes (the meters) change and that these changes occur independently for each staff, so that each staff carries it's own sequence of measures. Thus, the staff is the next level in the hierarchy. Finally there's the piano staff, which is composed of the right hand and left hand staves.

In what follows we will model this structure in this order (*cell*, measure, staff, piano staff), from bottom to top.

6.1 The cell

Before plunging into the code, observe the following characteristic of the *cell*:

1. It is composed of two layers: the top one which is an octave "chord" and the bottom one which is a straight eighth note run.
2. The total duration of the *cell* can vary, and is always the sum of the eight note funs.
3. The eight note runs are always stem down while the octave "chord" is always stem up.
4. The eight note runs are always beamed together and slurred, and the first two notes always have the dynamic markings 'f' 'p'.

The two "layers" of the *cell* we will model with two Voices inside a parallel Container. The top Voice will hold the octave "chord" while the lower Voice will hold the eighth note run. First the eighth notes:

```
>>> pitches = [1,2,3]
>>> notes = notetools.make_notes(pitches, [(1, 8)])
>>> beamtools.BeamSpanner(notes)
BeamSpanner(cs'8, d'8, ef'8)
>>> spannertools.SlurSpanner(notes)
SlurSpanner(cs'8, d'8, ef'8)
>>> contexttools.DynamicMark('f')(notes[0])
DynamicMark('f')(cs'8)
>>> contexttools.DynamicMark('p')(notes[1])
DynamicMark('p')(d'8)
```

```
>>> voice_lower = Voice(notes)
>>> voice_lower.name = 'rh_lower'
>>> marktools.LilyPondCommandMark('voiceTwo')(voice_lower)
LilyPondCommandMark('voiceTwo')(Voice-"rh_lower"{3})
```

The notes belonging to the eighth note run are first beamed and slurred. Then we add the dynamic marks to the first two notes, and finally we put them inside a Voice. After naming the voice we number it 2 so that the stems of the notes point down.

Now we construct the octave:

```
>>> import math
>>> n = int(math.ceil(len(pitches) / 2.))
>>> chord = Chord([pitches[0], pitches[0] + 12], (n, 8))
>>> marktools.Articulation('>')(chord)
Articulation('>')(<cs' cs''>4)
```

```
>>> voice_higher = Voice([chord])
>>> voice_higher.name = 'rh_higher'
>>> marktools.LilyPondCommandMark('voiceOne')(voice_higher)
LilyPondCommandMark('voiceOne')(Voice-"rh_higher"{1})
```

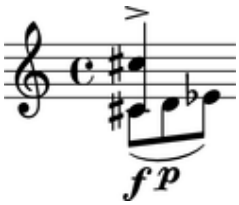
The duration of the chord is half the duration of the running eighth notes if the duration of the running notes is divisible by two. Otherwise the duration of the chord is the next integer greater than this half. We add the articulation marking and finally ad the Chord to a Voice, to which we set the number to 1, forcing the stem to always point up.

Finally we combine the two voices in a parallel Container:

```
>>> container = Container([voice_lower, voice_higher])
>>> container.is_parallel = True
```

This results in the complete *Désordre cell*:

```
>>> cell = Staff([container])
>>> show(cell)
```



Because this *cell* appears over and over again, we want to reuse this code to generate any number of these *cells*. We here encapsulate it in a function that will take only a list of pitches:

```
def make_desordre_cell(pitches):
    '''The function constructs and returns a *Désordre cell*.
    `pitches` is a list of numbers or, more generally, pitch tokens.
    '''
    notes = [Note(pitch, (1, 8)) for pitch in pitches]
    beamtools.BeamSpanner(notes)
    spannertools.SlurSpanner(notes)
    contexttools.DynamicMark('f')(notes[0])
    contexttools.DynamicMark('p')(notes[1])

    # make the lower voice
    lower_voice = Voice(notes)
    lower_voice.name = 'RH Lower Voice'
    marktools.LilyPondCommandMark('voiceTwo')(lower_voice)
    n = int(math.ceil(len(pitches) / 2.))
    chord = Chord([pitches[0], pitches[0] + 12], (n, 8))
    marktools.Articulation('>')(chord)

    # make the upper voice
    upper_voice = Voice([chord])
    upper_voice.name = 'RH Upper Voice'
    marktools.LilyPondCommandMark('voiceOne')(upper_voice)

    # combine them together
    container = Container([lower_voice, upper_voice])
    container.is_parallel = True

    # make all 1/8 beats breakable
    for leaf in lower_voice.leaves[:-1]:
        marktools.BarLine('')(leaf)

    return container
```

Now we can call this function to create any number of *cells*. That was actually the hardest part of reconstructing the opening of Ligeti's *Désordre*. Because the repetition of patters occurs also at the level of measures and staves, we will now define functions to create these other higher level constructs.

6.2 The measure

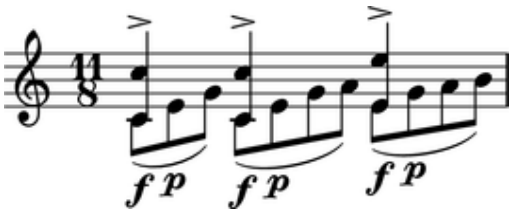
We define a function to create a measure from a list of lists of numbers:

```
def make_desordre_measure(pitches):
    '''Constructs a measure composed of *Désordre cells*.
    `pitches` is a list of lists of number (e.g., [[1, 2, 3], [2, 3, 4]])
    The function returns a DynamicMeasure.
    '''
    measure = measuretools.DynamicMeasure([ ])
    for sequence in pitches:
        measure.append(make_desordre_cell(sequence))

    # make denominator 8
    if contexttools.get_effective_time_signature(measure).denominator == 1:
        measure.denominator = 8

    return measure
```

The function is very simple. It simply creates a DynamicMeasure and then populates it with *cells* that are created internally with the function previously defined. The function takes a list *pitches* which is actually a list of lists

of pitches (e.g., `[[1,2,3], [2,3,4]]`. The list of lists of pitches is iterated to create each of the *cells* to be appended to the DynamicMeasures. We could have defined the function to take ready made *cells* directly, but we are building the hierarchy of functions so that we can pass simple lists of lists of numbers to generate the full structure. To construct a Ligeti measure we would call the function like so:

```
>>> pitches = [[0, 4, 7], [0, 4, 7, 9], [4, 7, 9, 11]]
>>> measure = make_desordre_measure(pitches)
>>> staff = Staff([measure])
>>> show(staff)
```



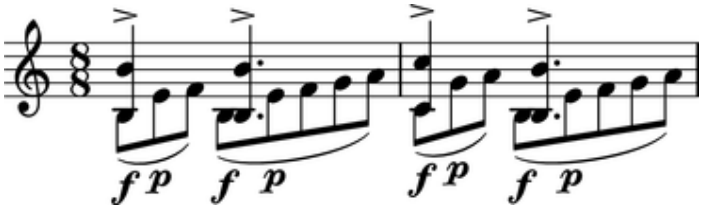
6.3 The staff

Now we move up to the next level, the staff:

```
def make_desordre_staff(pitches):
    staff = Staff()

    for sequence in pitches:
        measure = make_desordre_measure(sequence)
        staff.append(measure)

    return staff
```

The function again takes a plain list as argument. The list must be a list of lists (for measures) of lists (for cells) of pitches. The function simply constructs the Ligeti measures internally by calling our previously defined function and puts them inside a Staff. As with measures, we can now create full measure sequences with this new function:

```
>>> pitches = [[[-1, 4, 5], [-1, 4, 5, 7, 9]], [[0, 7, 9], [-1, 4, 5, 7, 9]]]
>>> staff = make_desordre_staff(pitches)
>>> show(staff)
```



6.4 The score

Finally a function that will generate the whole opening section of the piece *Désordre*:

```
def make_desordre_score(pitches):
    '''Returns a complete PianoStaff with Ligeti music!'''

    assert len(pitches) == 2
    piano_staff = scoretools.PianoStaff()

    # build the music...
    for hand in pitches:
        staff = make_desordre_staff(hand)
        piano_staff.append(staff)

    # set clef and key signature to left hand staff...
```

```
    contexttools.ClefMark('bass')(piano_staff[1])
    contexttools.KeySignatureMark('b', 'major')(piano_staff[1])

    # wrap the piano staff in a score, and return
    score = Score([piano_staff])

    return score
```

The function creates a PianoStaff, constructs Staves with Ligeti music and appends these to the empty PianoStaff. Finally it sets the clef and key signature of the lower staff to match the original score. The argument of the function is a list of length 2, depth 3. The first element in the list corresponds to the upper staff, the second to the lower staff.

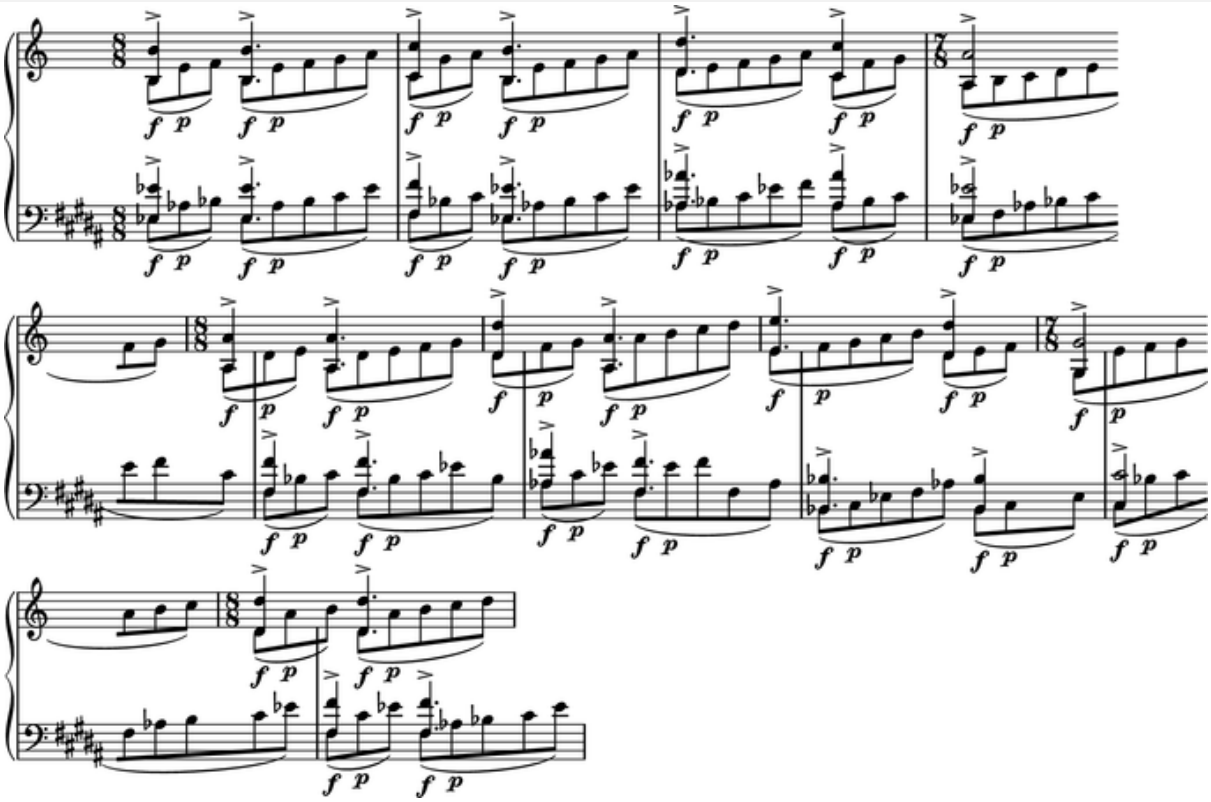
The final result:

```
>>> top = [[[-1, 4, 5], [-1, 4, 5, 7, 9]], [[0, 7, 9], [-1, 4, 5, 7, 9]], [[2, 4, 5, 7, 9], [0, 5, 7]], [[-3,
>>> bottom = [[[-9, -4, -2], [-9, -4, -2, 1, 3]], [[-6, -2, 1], [-9, -4, -2, 1, 3]], [[-4, -2, 1, 3, 6], [-4,
```

```
>>> score = make_desordre_score([top, bottom])
```

```
>>> from abjad.tools import documentationtools
>>> lilypond_file = documentationtools.make_ligeti_example_lilypond_file(score)
```

```
>>> show(lilypond_file)
```



Now that we have the redundant aspect of the piece compactly expressed and encapsulated, we can play around with it by changing the sequence of pitches.

In order for each staff to carry its own sequence of independent measure changes, LilyPond requires some special setting up prior to rendering. Specifically, one must move the LilyPond `Timing_translator` out from the score context and into the staff context.

(You can refer to the LilyPond documentation on Polymetric notation to learn all about how this works.)

In this example we a custom `documentationtools` function to set up our LilyPond file automatically.

CHAPTER

SEVEN

MOZART: *MUSIKALISCHES WÜRFELSPIEL*

Note: Explore the *abjad/demos/mozart/* directory for the complete code to this example, or import it into your Python session directly with:

- *from abjad.demos import mozart*

Mozart's dice game is a method for aleatorically generating sixteen-measure-long minuets. For each measure, two six-sided dice are rolled, and the sum of the dice used to look up a measure number in one of two tables (one for each half of the minuet). The measure number then locates a single measure from a collection of musical fragments. The fragments are concatenated together, and "music" results.

Implementing the dice game in a composition environment is somewhat akin to (although also somewhat more complicated than) the ubiquitous hello world program which every programming language uses to demonstrate its basic syntax.

Note: The musical dice game in question (*k516f*) has long been attributed to Mozart, albeit inconclusively. Its actual provenance is a musicological problem with which we are unconcerned here.

7.1 The materials

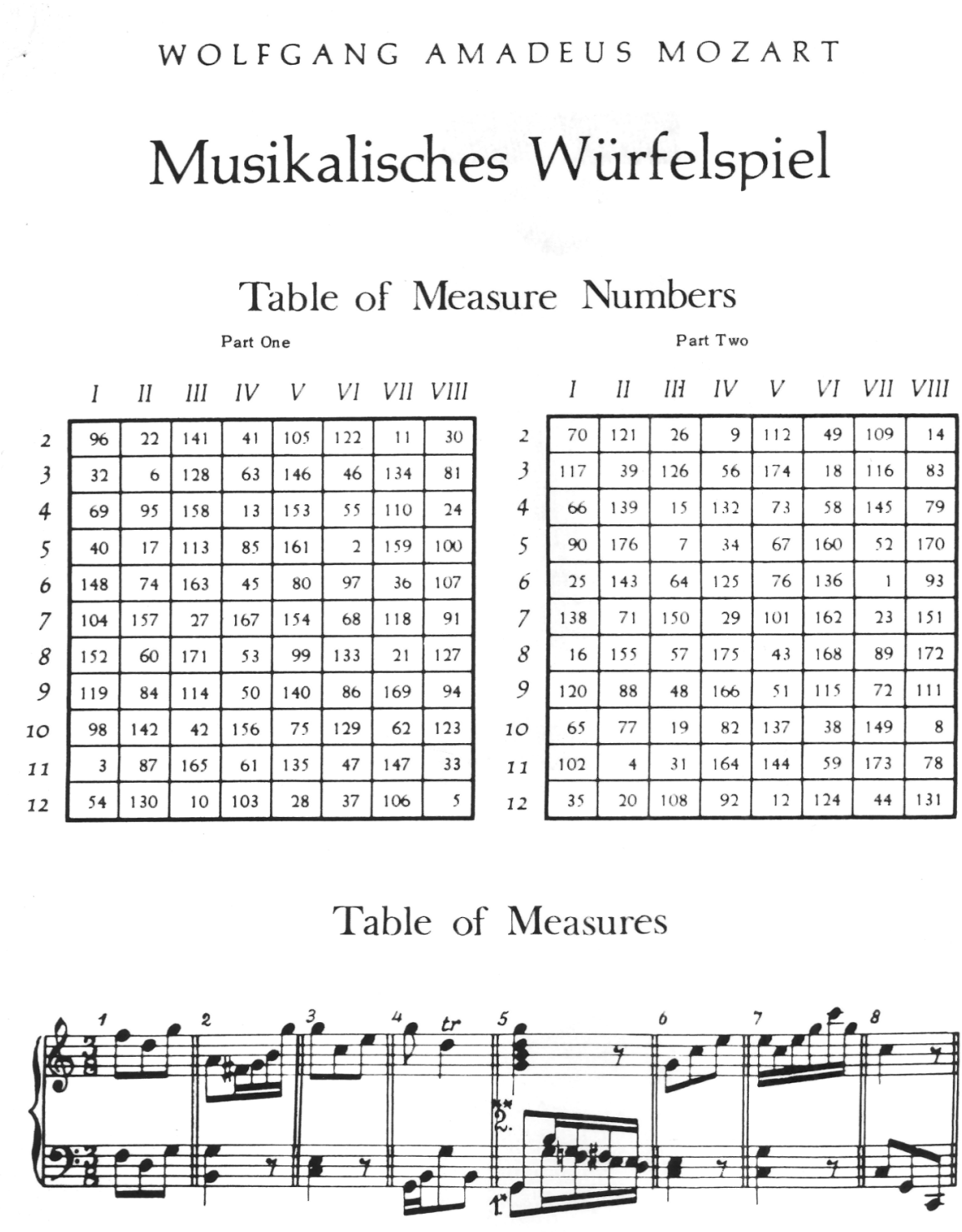
At the heart of the dice game is a large collection, *or corpus*, of musical fragments. Each fragment is a single 3/8 measure, consisting of a treble voice and a bass voice. Traditionally, these fragments are stored in a "score", or "table of measures", and located via two tables of measure numbers, which act as lookups, indexing into that collection.

Duplicate measures in the original corpus are common. Notably, the 8th measure - actually a pair of measures represent the first and second alternate ending of the first half of the minuet - are always identical. The last measure of the piece is similarly limited - there are only two possibilities rather than the usual eleven (for the numbers 2 to 12, being all the possible sums of two 6-sided dice).

How might we store this corpus compactly?

Some basic musical information in Abjad can be stored as strings, rather than actual collections of class instances. Abjad can parse simple LilyPond strings via `p`, which interprets a subset of LilyPond syntax, and understands basic concepts like notes, chords, rests and skips, as well as beams, slurs, ties, and articulations.

```
>>> lily_string = r"\new Staff { c'4 ( d'4 <cs' e'>8 ) -. r8 <g' b' d''>4 ^ \marcato ~ <g' b' d''>1 }"
>>> parsed_result = p(lily_string)
>>> f(parsed_result)
\new Staff {
    c'4 (
    d'4
    <cs' e'>8 -\staccato )
```



WOLFGANG AMADEUS MOZART

Musikalisches Würfelspiel

Table of Measure Numbers

Part One

	I	II	III	IV	V	VI	VII	VIII
2	96	22	141	41	105	122	11	30
3	32	6	128	63	146	46	134	81
4	69	95	158	13	153	55	110	24
5	40	17	113	85	161	2	159	100
6	148	74	163	45	80	97	36	107
7	104	157	27	167	154	68	118	91
8	152	60	171	53	99	133	21	127
9	119	84	114	50	140	86	169	94
10	98	142	42	156	75	129	62	123
11	3	87	165	61	135	47	147	33
12	54	130	10	103	28	37	106	5

Part Two

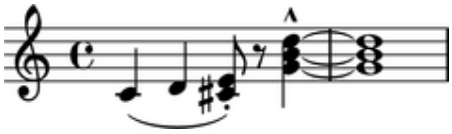
	I	II	III	IV	V	VI	VII	VIII
2	70	121	26	9	112	49	109	14
3	117	39	126	56	174	18	116	83
4	66	139	15	132	73	58	145	79
5	90	176	7	34	67	160	52	170
6	25	143	64	125	76	136	1	93
7	138	71	150	29	101	162	23	151
8	16	155	57	175	43	168	89	172
9	120	88	48	166	51	115	72	111
10	65	77	19	82	137	38	149	8
11	102	4	31	164	144	59	173	78
12	35	20	108	92	12	124	44	131

Table of Measures

Figure 7.1: *Part of a pen-and-paper implementation from the 20th century.*

```
    r8
    <g' b' d''>4 ^\marcato ~
    <g' b' d''>1
}
```

```
>>> show(parsed_result)
```



So, instead of storing our musical information as Abjad components, we'll represent each fragment in the corpus as a pair of strings: one representing the bass voice contents, and the other representing the treble. This pair of strings can be packaged together into a collection. For this implementation, we'll package them into a dictionary. Python dictionaries are cheap, and often provide more clarity than lists; the composer does not have to rely on remembering a convention for what data should appear in which position in a list - they can simply label that data semantically. In our musical dictionary, the treble voice will use the key 't' and the bass voice will use the key 'b'.

```
>>> fragment = {'t': "g''8 ( e''8 c''8 )", 'b': '<c e>4 r8'}
```

Instead of relying on measure number tables to find our fragments - as in the original implementation, we'll package our fragment dictionaries into a list of lists of fragment dictionaries. That is to say, each of the sixteen measures in the piece will be represented by a list of fragment dictionaries. Furthermore, the 8th measure, which breaks the pattern, will simply be a list of two fragment dictionaries. Structuring our information in this way lets us avoid using measure number tables entirely; Python's list-indexing affordances will take care of that for us. The complete corpus looks like this:

```
def make_mozart_measure_corpus():
    return [
        [
            {'b': 'c4 r8', 't': "e''8 c''8 g'8"},
            {'b': '<c e>4 r8', 't': "g'8 c''8 e''8"},
            {'b': '<c e>4 r8', 't': "g''8 ( e''8 c''8 )"},
            {'b': '<c e>4 r8', 't': "c''16 b'16 c''16 e''16 g'16 c''16"},
            {'b': '<c e>4 r8', 't': "c'''16 b''16 c'''16 g''16 e''16 c''16"},
            {'b': 'c4 r8', 't': "e''16 d''16 e''16 g''16 c'''16 g''16"},
            {'b': '<c e>4 r8', 't': "g''8 f''16 e''16 d''16 c''16"},
            {'b': '<c e>4 r8', 't': "e''16 c''16 g''16 e''16 c'''16 g''16"},
            {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "c''8 g'8 e''8"},
            {'b': '<c e>4 r8', 't': "g''8 c''8 e''8"},
            {'b': 'c8 c8 c8', 't': "<e' c''>8 <e' c''>8 <e' c''>8"},
        ],
        [
            {'b': 'c4 r8', 't': "e''8 c''8 g'8"},
            {'b': '<c e>4 r8', 't': "g'8 c''8 e''8"},
            {'b': '<c e>4 r8', 't': "g''8 e''8 c''8"},
            {'b': '<e g>4 r8', 't': "c''16 g'16 c''16 e''16 g'16 c''16"},
            {'b': '<c e>4 r8', 't': "c'''16 b''16 c'''16 g''16 e''16 c''16"},
            {'b': 'c4 r8', 't': "e''16 d''16 e''16 g''16 c'''16 g''16"},
            {'b': '<c e>4 r8', 't': "g''8 f''16 e''16 d''16 c''16"},
            {'b': '<c e>4 r8', 't': "c''16 g'16 e''16 c''16 g''16 e''16"},
            {'b': '<c e>4 r8', 't': "c''8 g'8 e''8"},
            {'b': '<c e>4 <c g>8', 't': "g''8 c''8 e''8"},
            {'b': 'c8 c8 c8', 't': "<e' c''>8 <e' c''>8 <e' c''>8"},
        ],
        [
            {'b': '<b, g>4 g,8', 't': "d''16 e''16 f''16 d''16 c''16 b'16"},
            {'b': 'g,4 r8', 't': "b'8 d''8 g''8"},
            {'b': 'g,4 r8', 't': "b'8 d''16 b'16 a'16 g'16"},
            {'b': '<g b>4 r8', 't': "f''8 d''8 b'8"},
            {'b': '<b, d>4 r8', 't': "g''16 fs''16 g''16 d''16 b'16 g'16"},
            {'b': '<g b>4 r8', 't': "f''16 e''16 f''16 d''16 c''16 b'16"},
            {'b': '<g, g>4 <b, g>8', 't': "b'16 c''16 d''16 e''16 f''16 d''16"},
            {'b': 'g8 g8 g8', 't': "<b' d''>8 <b' d''>8 <b' d''>8"},
            {'b': 'g,4 r8', 't': "b'16 c''16 d''16 b'16 a'16 g'16"},
            {'b': 'b,4 r8', 't': "d''8 ( b'8 g'8 )"},
            {'b': 'g4 r8', 't': "b'16 a'16 b'16 c''16 d''16 b'16"},
        ],
```

```
        [
            {'b': '<c e>4 r8', 't': "c''16 b'16 c''16 e''16 g'8"},
            {'b': 'c4 r8', 't': "e''16 c''16 b'16 c''16 g'8"},
            {'b': '<e g>4 r8', 't': "c''8 ( g'8 e'8 )"},
            {'b': '<e g>4 r8', 't': "c''8 e''8 g'8"},
            {'b': '<e g>4 r8', 't': "c''16 b'16 c''16 g'16 e'16 c'16"},
            {'b': '<c e>4 r8', 't': "c''8 c''16 d''16 e''8"},
            {'b': 'c4 r8', 't': "<c'' e''>8 <c'' e''>16 <d'' f''>16 <e'' g''>8"},
            {'b': '<e g>4 r8', 't': "c''8 e''16 c''16 g'8"},
            {'b': '<e g>4 r8', 't': "c''16 g'16 e''16 c''16 g''8"},
            {'b': '<e g>4 r8', 't': "c''8 e''16 c''16 g''8"},
            {'b': '<e g>4 r8', 't': "c''16 e''16 c''16 g'16 e'8"},
        ],
        [
            {'b': 'c4 r8', 't': "fs''8 a''16 fs''16 d''16 fs''16"},
            {'b': 'c8 c8 c8', 't': "<fs' d''>8 <d'' fs''>8 <fs'' a''>8"},
            {'b': 'c4 r8', 't': "d''16 a'16 fs''16 d''16 a''16 fs''16"},
            {'b': 'c8 c8 c8', 't': "<fs' d''>8 <fs' d''>8 <fs' d''>8"},
            {'b': 'c4 r8', 't': "d''8 a'8 ^\\turn fs''8"},
            {'b': 'c4 r8', 't': "d''16 cs''16 d''16 fs''16 a''16 fs''16"},
            {'b': '<c a>4 <c a>8', 't': "fs''8 a''8 d''8"},
            {'b': '<c fs>8 <c fs>8 <c a>8', 't': "a'8 a'16 d''16 fs''8"},
            {'b': 'c8 c8 c8', 't': "<d'' fs''>8 <d'' fs''>8 <d'' fs''>8"},
            {'b': '<c d>8 <c d>8 <c d>8', 't': "fs''8 fs''16 d''16 a''8"},
            {'b': '<c a>4 r8', 't': "fs''16 d''16 a'16 a''16 fs''16 d''16"},
        ],
        [
            {'b': '<b, d>8 <b, d>8 <b, d>8', 't': "g''16 fs''16 g''16 b''16 d''8"},
            {'b': '<b, d>4 r8', 't': "g''8 b''16 g''16 d''16 b'16"},
            {'b': '<b, d>4 r8', 't': "g''8 b''8 d''8"},
            {'b': '<b, g>4 r8', 't': "a'8 fs'16 g'16 b'16 g''16"},
            {'b': '<b, d>4 <b, g>8', 't': "g''16 fs''16 g''16 d''16 b'16 g'16"},
            {'b': 'b,4 r8', 't': "g''8 b''16 g''16 d''16 g''16"},
            {'b': '<b, g>4 r8', 't': "d''8 g''16 d''16 b'16 d''16"},
            {'b': '<b, g>4 r8', 't': "d''8 d''16 g''16 b''8"},
            {'b': '<b, d>8 <b, d>8 <b, g>8', 't': "a''16 g''16 fs''16 g''16 d''8"},
            {'b': '<b, d>4 r8', 't': "g''8 g''16 d''16 b''8"},
            {'b': '<b, d>4 r8', 't': "g''16 b''16 g''16 d''16 b'8"},
        ],
        [
            {'b': 'c8 d8 d,8', 't': "e''16 c''16 b'16 a'16 g'16 fs'16"},
            {'b': 'c8 d8 d,8', 't': "a'16 e''16 <b' d''>16 <a' c''>16 <g' b'>16 <fs' a'>16"},
            {'b': 'c8 d8 d,8', 't': "<b' d''>16 ( <a' c''>16 ) <a' c''>16 ( <g' b'>16 ) <g' b'>16 ( <fs' a'>16
            {'b': 'c8 d8 d,8', 't': "e''16 g''16 d''16 c''16 b'16 a'16"},
            {'b': 'c8 d8 d,8', 't': "a'16 e''16 d''16 g''16 fs''16 a''16"},
            {'b': 'c8 d8 d,8', 't': "e''16 a''16 g''16 b''16 fs''16 a''16"},
            {'b': 'c8 d8 d,8', 't': "c''16 e''16 g''16 d''16 a'16 fs''16"},
            {'b': 'c8 d8 d,8', 't': "e''16 g''16 d''16 g''16 a'16 fs''16"},
            {'b': 'c8 d8 d,8', 't': "e''16 c''16 b'16 g'16 a'16 fs'16"},
            {'b': 'c8 d8 d,8', 't': "e''16 c'''16 b''16 g''16 a''16 fs''16"},
            {'b': 'c8 d8 d,8', 't': "a'8 d''16 c''16 b'16 a'16"},
        ],
        [
            {'b': 'g,8 g16 f16 e16 d16', 't': "<g' b' d'' g''>4 r8"},
            {'b': 'g,8 b16 g16 fs16 e16', 't': "<g' b' d'' g''>4 r8"},
        ],
        [
            {'b': 'd4 c8', 't': "fs''8 a''16 fs''16 d''16 fs''16"},
            {'b': '<d fs>4 r8', 't': "d''16 a'16 d''16 fs''16 a''16 fs''16"},
            {'b': '<d a>8 <d fs>8 <c d>8', 't': "fs''8 a''8 fs''8"},
            {'b': '<c a>4 <c a>8', 't': "fs''16 a''16 d'''16 a''16 fs''16 a''16"},
            {'b': 'd4 c8', 't': "d'16 fs'16 a'16 d''16 fs''16 a''16"},
            {'b': 'd,16 d16 cs16 d16 c16 d16', 't': "<a' d'' fs''>8 fs''4 ^\\trill"},
            {'b': '<d fs>4 <c fs>8', 't': "a''8 ( fs''8 d''8 )"},
            {'b': '<d fs>4 <c fs>8', 't': "d'''8 a''16 fs''16 d''16 a'16"},
            {'b': '<d fs>4 r8', 't': "d''16 a'16 d''8 fs''8"},
            {'b': '<c a>4 <c a>8', 't': "fs''16 d''16 a'8 fs''8"},
            {'b': '<d fs>4 <c a>8', 't': "a'8 d''8 fs''8"},
        ],
        [
            {'b': '<b, g>4 r8', 't': "g''8 b''16 g''16 d''8"},
            {'b': 'b,16 d16 g16 d16 b,16 g,16', 't': "g''8 g'8 g'8"},
            {'b': 'b,4 r8', 't': "g''16 b''16 g''16 b''16 d''8"},
```

```
            {'b': '<b, d>4 <b, d>8', 't': "a''16 g''16 b''16 g''16 d''16 g''16"},
            {'b': '<b, d>4 <b, d>8', 't': "g''8 d''16 b'16 g'8"},
            {'b': '<b, d>4 <b, d>8', 't': "g''16 b''16 d'''16 b''16 g''8"},
            {'b': '<b, d>4 r8', 't': "g''16 b''16 g''16 d''16 b'16 g'16"},
            {'b': '<b, d>4 <b, d>8', 't': "g''16 d''16 g''16 b''16 g''16 d''16"},
            {'b': '<b, d>4 <b, g>8', 't': "g''16 b''16 g''8 d''8"},
            {'b': 'g,16 b,16 g8 b,8', 't': "g''8 d''4 ^\\trill"},
            {'b': 'b,4 r8', 't': "g''8 b''16 d'''16 d''8"},
        ],
        [
            {'b': "c16 e16 g16 e16 c'16 c16", 't': "<c'' e''>8 <c'' e''>8 <c'' e''>8"},
            {'b': 'e4 e16 c16', 't': "c''16 g'16 c''16 e''16 g''16 <c'' e''>16"},
            {'b': '<c g>4 <c e>8', 't': "e''8 g''16 e''16 c''8"},
            {'b': '<c g>4 r8', 't': "e''16 c''16 e''16 g''16 c'''16 g''16"},
            {'b': '<c g>4 <c g>8', 't': "e''16 g''16 c'''16 g''16 e''16 c''16"},
            {'b': 'c16 b,16 c16 d16 e16 fs16', 't': "<g' c'' e''>8 e''4 ^\\trill"},
            {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "e''8 c''8 g'8"},
            {'b': '<c g>4 <c e>8', 't': "e''8 c''16 e''16 g''16 c'''16"},
            {'b': '<c g>4 <c e>8', 't': "e''16 c''16 e''8 g''8"},
            {'b': '<c g>4 <c g>8', 't': "e''16 c''16 g'8 e''8"},
            {'b': '<c g>4 <c e>8', 't': "e''8 ( g''8 c'''8 )"},
        ],
        [
            {'b': 'g4 g,8', 't': "<c'' e''>8 <b' d''>8 r8"},
            {'b': '<g, g>4 g8', 't': "d''16 b'16 g'8 r8"},
            {'b': 'g8 g,8 r8', 't': "<c'' e''>8 <b' d''>16 <g' b'>16 g'8"},
            {'b': 'g4 r8', 't': "e''16 c''16 d''16 b'16 g'8"},
            {'b': 'g8 g,8 r8', 't': "g''16 e''16 d''16 b'16 g'8"},
            {'b': 'g4 g,8', 't': "b'16 d''16 g''16 d''16 b'8"},
            {'b': 'g8 g,8 r8', 't': "e''16 c''16 b'16 d''16 g''8"},
            {'b': '<g b>4 r8', 't': "d''16 b''16 g''16 d''16 b'8"},
            {'b': '<b, g>4 <b, d>8', 't': "d''16 b'16 g'8 g''8"},
            {'b': 'g16 fs16 g16 d16 b,16 g,16', 't': "d''8 g'4"},
        ],
        [
            {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "e''8 c''8 g'8"},
            {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "g'8 c''8 e''8"},
            {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "g''8 e''8 c''8"},
            {'b': '<c e>4 <e g>8', 't': "c''16 b'16 c''16 e''16 g'16 c''16"},
            {'b': '<c e>4 <c g>8', 't': "c'''16 b''16 c'''16 g''16 e''16 c''16"},
            {'b': '<c g>4 <c e>8', 't': "e''16 d''16 e''16 g''16 c'''16 g''16"},
            {'b': '<c e>4 r8', 't': "g''8 f''16 e''16 d''16 c''16"},
            {'b': '<c e>4 r8', 't': "c''16 g'16 e''16 c''16 g''16 e''16"},
            {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "c''8 g'8 e''8"},
            {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "g''8 c''8 e''8"},
            {'b': 'c8 c8 c8', 't': "<e' c''>8 <e' c''>8 <e' c''>8"},
        ],
        [
            {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "e''8 ( c''8 g'8 )"},
            {'b': '<c e>4 <c g>8', 't': "g'8 ( c''8 e''8 )"},
            {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "g''8 e''8 c''8"},
            {'b': '<c e>4 <c e>8', 't': "c''16 b'16 c''16 e''16 g'16 c''16"},
            {'b': '<c e>4 r8', 't': "c'''16 b''16 c'''16 g''16 e''16 c''16"},
            {'b': '<c g>4 <c e>8', 't': "e''16 d''16 e''16 g''16 c'''16 g''16"},
            {'b': '<c e>4 <e g>8', 't': "g''8 f''16 e''16 d''16 c''16"},
            {'b': '<c e>4 r8', 't': "c''16 g'16 e''16 c''16 g''16 e''16"},
            {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "c''8 g'8 e''8"},
            {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "g''8 c''8 e''8"},
            {'b': 'c8 c8 c8', 't': "<e' c''>8 <e' c''>8 <e' c''>8"},
        ],
        [
            {'b': "<f a>4 <g d'>8", 't': "d''16 f''16 d''16 f''16 b'16 d''16"},
            {'b': 'f4 g8', 't': "d''16 f''16 a''16 f''16 d''16 b'16"},
            {'b': 'f4 g8', 't': "d''16 f''16 a'16 d''16 b'16 d''16"},
            {'b': 'f4 g8', 't': "d''16 ( cs''16 ) d''16 f''16 g'16 b'16"},
            {'b': 'f8 d8 g8', 't': "f''8 d''8 g''8"},
            {'b': 'f16 e16 d16 e16 f16 g16', 't': "f''16 e''16 d''16 e''16 f''16 g''16"},
            {'b': 'f16 e16 d8 g8', 't': "f''16 e''16 d''8 g''8"},
            {'b': 'f4 g8', 't': "f''16 e''16 d''16 c''16 b'16 d''16"},
            {'b': 'f4 g8', 't': "f''16 d''16 a'8 b'8"},
            {'b': 'f4 g8', 't': "f''16 a''16 a'8 b'16 d''16"},
            {'b': 'f4 g8', 't': "a'8 f''16 d''16 a'16 b'16"},
        ],
```

```
        [
            {'b': 'c8 g,8 c,8', 't': "c''4 r8"},
            {'b': 'c4 c,8', 't': "c''8 c'8 r8"},
        ],
    ]
```

We can then use the `p()` function we saw earlier to "build" the treble and bass components of a measure like this:

```
def make_mozart_measure(measure_dict):
    # parse the contents of a measure definition dictionary
    # wrap the expression to be parsed inside a LilyPond { } block
    treble = p('{{ {} }}'.format(measure_dict['t']))
    bass = p('{{ {} }}'.format(measure_dict['b']))
    return treble, bass
```

Let's try with a measure-definition of our own:

```
>>> my_measure_dict = {'b': r'c4 ^\trill r8', 't': "e''8 ( c''8 g'8 )"}
>>> treble, bass = make_mozart_measure(my_measure_dict)
```

```
>>> f(treble)
{
    e''8 (
    c''8
    g'8 )
}
```

```
>>> f(bass)
{
    c4 ^\trill
    r8
}
```

Now with one from the Mozart measure collection defined earlier. We'll grab the very last choice for the very last measure:

```
>>> my_measure_dict = make_mozart_measure_corpus()[-1][-1]
>>> treble, bass = make_mozart_measure(my_measure_dict)
```

```
>>> f(treble)
{
    c''8
    c'8
    r8
}
```

```
>>> f(bass)
{
    c4
    c,8
}
```

7.2 The structure

After storing all of the musical fragments into a corpus, concatenating those elements into a musical structure is relatively trivial. We'll use the `choice()` function from Python's *random* module. `random.choice()` randomly selects one element from an input list.

```
>>> import random
>>> my_list = [1, 'b', 3]
>>> my_result = [random.choice(my_list) for i in range(20)]
>>> my_result
[3, 3, 'b', 1, 'b', 'b', 3, 1, 'b', 'b', 3, 'b', 1, 3, 'b', 1, 3, 3, 3, 3]
```

Our corpus is a list comprising sixteen sublists, one for each measure in the minuet. To build our musical structure, we can simply iterate through the corpus and call *choice* on each sublist, appending the chosen results to another

list. The only catch is that the *eighth* measure of our minuet is actually the first-and-second-ending for the repeat of the first phrase. The sublist of the corpus for measure eight contains *only* the first and second ending definitions, and both of those measures should appear in the final piece, always in the same order. We'll have to intercept that sublist while we iterate through the corpus and apply some different logic.

The easist way to intercept measure eight is to use the Python builtin *enumerate*, which allows you to iterate through a collection while also getting the index of each element in that collection:

```
def choose_mozart_measures():
    measure_corpus = make_mozart_measure_corpus()
    chosen_measures = []
    for i, choices in enumerate(measure_corpus):
        if i == 7: # get both alternative endings for mm. 8
            chosen_measures.extend(choices)
        else:
            choice = random.choice(choices)
            chosen_measures.append(choice)
    return chosen_measures
```

Note: In *choose_mozart_measures* we test for index *7*, rather then *8*, because list indices count from *0* instead of *1*.

The result will be a *seventeen*-item-long list of measure definitions:

```
>>> choices = choose_mozart_measures()
>>> for i, measure in enumerate(choices):
...     print i, measure
...
0 {'b': '<c e>4 r8', 't': "c''16 b'16 c''16 e''16 g'16 c''16"}
1 {'b': '<c e>4 r8', 't': "c''8 g'8 e''8"}
2 {'b': 'b,4 r8', 't': "d''8 ( b'8 g'8 )"}
3 {'b': '<e g>4 r8', 't': "c''8 e''16 c''16 g'8"}
4 {'b': 'c4 r8', 't': "d''16 cs''16 d''16 fs''16 a''16 fs''16"}
5 {'b': '<b, d>4 r8', 't': "g''8 b''16 g''16 d''16 b'16"}
6 {'b': 'c8 d8 d,8', 't': "a'16 e''16 d''16 g''16 fs''16 a''16"}
7 {'b': 'g,8 g16 f16 e16 d16', 't': "<g' b' d'' g''>4 r8"}
8 {'b': 'g,8 b16 g16 fs16 e16', 't': "<g' b' d'' g''>4 r8"}
9 {'b': '<d fs>4 <c fs>8', 't': "a''8 ( fs''8 d''8 )"}
10 {'b': 'b,4 r8', 't': "g''8 b''16 d'''16 d''8"}
11 {'b': '<c g>4 <c e>8', 't': "e''8 ( g''8 c'''8 )"}
12 {'b': 'g8 g,8 r8', 't': "g''16 e''16 d''16 b'16 g'8"}
13 {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "g''8 c''8 e''8"}
14 {'b': '<c e>16 g16 <c e>16 g16 <c e>16 g16', 't': "g''8 e''8 c''8"}
15 {'b': 'f4 g8', 't': "f''16 d''16 a'8 b'8"}
16 {'b': 'c4 c,8', 't': "c''8 c'8 r8"}
```

7.3 The score

Now that we have our raw materials, and a way to organize them, we can start building our score. The tricky part here is figuring out how to implement LilyPond's repeat structure in Abjad. LilyPond structures its repeats something like this:

```
\repeat volta n {
    music to be repeated
}

\alternative {
    { ending 1 }
    { ending 2 }
    { ending n }
}

...music after the repeat...
```

What you see above is really just two containers, each with a little text ("repeat volta n" and "alternative") prepended to their opening curly brace. To create that structure in Abjad, we'll need to use the `LilyPondCommandMark` class, which allows you to place LilyPond commands like "break" relative to any score component:

```
>>> con = Container("c'4 d'4 e'4 f'4")
>>> mark = marktools.LilyPondCommandMark('before-the-container', 'before')(con)
>>> mark = marktools.LilyPondCommandMark('after-the-container', 'after')(con)
>>> mark = marktools.LilyPondCommandMark('opening-of-the-container', 'opening')(con)
>>> mark = marktools.LilyPondCommandMark('closing-of-the-container', 'closing')(con)
>>> mark = marktools.LilyPondCommandMark('to-the-right-of-a-note', 'right')(con[2])
>>> f(con)
\before-the-container
{
    \opening-of-the-container
    c'4
    d'4
    e'4 \to-the-right-of-a-note
    f'4
    \closing-of-the-container
}
\after-the-container
```

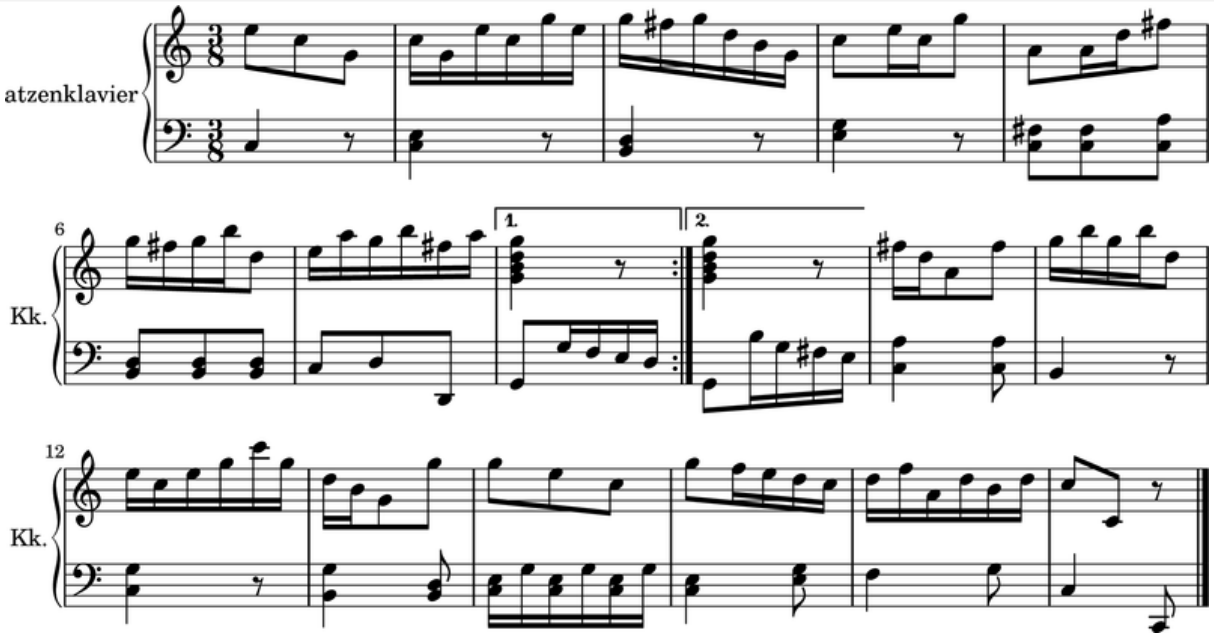
Notice the second argument to each `LilyPondCommandMark` above, like *before* and *closing*. These are format slot indications, which control where the command is placed in the LilyPond code relative to the score element it is attached to. To mimic LilyPond's repeat syntax, we'll have to create two `LilyPondCommandMark` instances, both using the "before" format slot, insuring that their command is placed before their container's opening curly brace.

Now let's take a look at the code that puts our score together:

```
def make_mozart_score():

    score_template = scoretemplatetools.TwoStaffPianoScoreTemplate()
    score = score_template()

    # select the measures to use
    choices = choose_mozart_measures()

    # create and populate the volta containers
    treble_volta = Container()
    bass_volta = Container()
    for choice in choices[:7]:
        treble, bass = make_mozart_measure(choice)
        treble_volta.append(treble)
        bass_volta.append(bass)

    # add marks to the volta containers
    marktools.LilyPondCommandMark(
        'repeat volta 2', 'before'
        )(treble_volta)
    marktools.LilyPondCommandMark(
        'repeat volta 2', 'before'
        )(bass_volta)

    # add the volta containers to our staves
    score['RH Voice'].append(treble_volta)
    score['LH Voice'].append(bass_volta)

    # create and populate the alternative ending containers
    treble_alternative = Container()
    bass_alternative = Container()
    for choice in choices[7:9]:
        treble, bass = make_mozart_measure(choice)
        treble_alternative.append(treble)
        bass_alternative.append(bass)

    # add marks to the alternative containers
    marktools.LilyPondCommandMark(
        'alternative', 'before'
        )(treble_alternative)
```

```
    marktools.LilyPondCommandMark(
        'alternative', 'before'
        )(bass_alternative)

    # add the alternative containers to our staves
    score['RH Voice'].append(treble_alternative)
    score['LH Voice'].append(bass_alternative)

    # create the remaining measures
    for choice in choices[9:]:
        treble, bass = make_mozart_measure(choice)
        score['RH Voice'].append(treble)
        score['LH Voice'].append(bass)

    # add marks
    contexttools.TimeSignatureMark((3, 8))(score['RH Staff'])
    marktools.BarLine('|.')(score['RH Voice'][-1])
    marktools.BarLine('|.')(score['LH Voice'][-1])

    # remove the old, default Piano InstrumentMark attached to the PianoStaff
    # and add a custom instrument mark
    contexttools.detach_instrument_marks_attached_to_component(score['Piano Staff'])
    contexttools.InstrumentMark(
        'Katzenklavier', 'kk.',
        target_context = scoretools.PianoStaff
        )(score['Piano Staff'])

    return score
```

```
>>> score = make_mozart_score()
>>> show(score)
```



Note: Our instrument name got cut off! Looks like we need to do a little formatting. Keep reading...

7.4 The document

As you can see above, we've now got our randomized minuet. However, we can still go a bit further. LilyPond provides a wide variety of settings for controlling the overall *look* of a musical document, often through its *header*, *layout* and *paper* blocks. Abjad, in turn, gives us object-oriented access to these settings through the its *lilypondfiletools* module.

We'll use `abjad.tools.lilypondfiletools.make_basic_lilypond_file()` to wrap our

`Score` inside a `LilyPondFile` instance. From there we can access the other "blocks" of our document to add a title, a composer's name, change the global staff size, paper size, staff spacing and so forth.

```
def make_mozart_lilypond_file():
    score = make_mozart_score()
    lily = lilypondfiletools.make_basic_lilypond_file(score)
    title = markuptools.Markup(r'\bold \sans "Ein Musikalisches Wuerfelspiel"')
    composer = schemetools.Scheme("W. A. Mozart (maybe?)")
    lily.global_staff_size = 12
    lily.header_block.title = title
    lily.header_block.composer = composer
    lily.layout_block.ragged_right = True
    lily.paper_block.markup_system_spacing__basic_distance = 8
    lily.paper_block.paper_width = 180
    return lily
```

```
>>> lilypond_file = make_mozart_lilypond_file()
>>> print lilypond_file
LilyPondFile(Score-"Two-Staff Piano Score"<<1>>)
```

```
>>> print lilypond_file.header_block
HeaderBlock(2)
```

```
>>> f(lilypond_file.header_block)
\header {
    composer = #"W. A. Mozart (maybe?)"
    title = \markup {
        \bold
            \sans
                "Ein Musikalisches Wuerfelspiel"
        }
}
```

```
>>> print lilypond_file.layout_block
LayoutBlock(1)
```

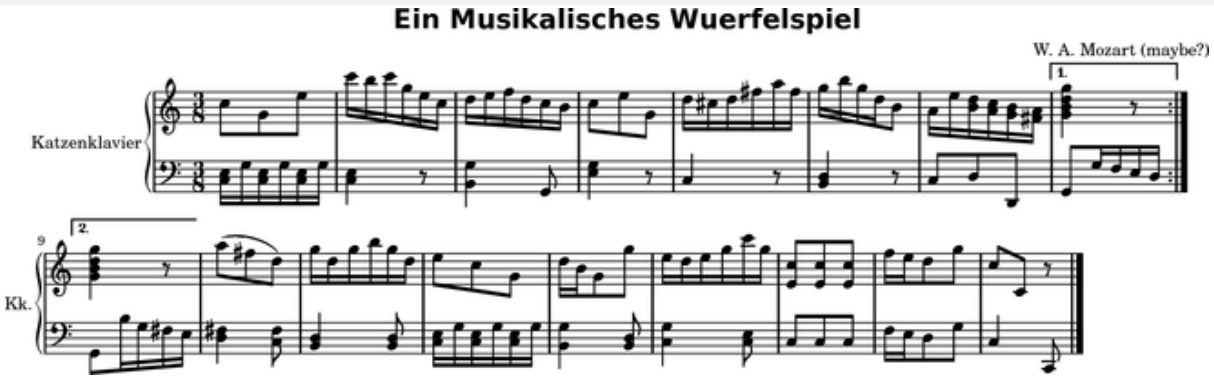
```
>>> f(lilypond_file.layout_block)
\layout {
    ragged-right = ##t
}
```

```
>>> print lilypond_file.paper_block
PaperBlock(2)
```

```
>>> f(lilypond_file.paper_block)
\paper {
    markup-system-spacing #'basic-distance = #8
    paper-width = #180
}
```

And now the final result:

```
>>> show(lilypond_file)
```



CHAPTER

EIGHT

PÄRT: *CANTUS IN MEMORY OF BENJAMIN BRITTEN*

Note: Explore the *abjad/demos/part/* directory for the complete code to this example, or import it into your Python session directly with:

- *from abjad.demos import part*

Let's make some imports:

```
>>> import copy
>>> from abjad import *
```

```
def make_part_lilypond_file():

    score_template = PartCantusScoreTemplate()
    score = score_template()

    add_bell_music_to_score(score)
    add_string_music_to_score(score)

    apply_bowing_marks(score)
    apply_dynamic_marks(score)
    apply_expressive_marks(score)
    apply_page_breaks(score)
    apply_rehearsal_marks(score)
    apply_final_bar_lines(score)

    configure_score(score)
    lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
    configure_lilypond_file(lilypond_file)

    return lilypond_file
```

8.1 The score template

```
class PartCantusScoreTemplate(scoretemplatetools.ScoreTemplate):

    ### INITIALIZER ###

    def __init__(self):
        pass

    ### SPECIAL METHODS ###

    def __call__(self):

        # make bell voice and staff
        bell_voice = voicetools.Voice(name='Bell Voice')
```

```
        bell_staff = stafftools.Staff([bell_voice], name='Bell Staff')
        contexttools.ClefMark('treble')(bell_staff)
        contexttools.InstrumentMark('Campana in La', 'Camp.')(bell_staff)
        contexttools.TempoMark((1, 4), (112, 120))(bell_staff)
        contexttools.TimeSignatureMark((6, 4))(bell_staff)

        # make first violin voice and staff
        first_violin_voice = voicetools.Voice(name='First Violin Voice')
        first_violin_staff = stafftools.Staff([first_violin_voice], name='First Violin Staff')
        contexttools.ClefMark('treble')(first_violin_staff)
        instrumenttools.Violin(
            instrument_name_markup='Violin I',
            short_instrument_name_markup='Vl. I'
            )(first_violin_staff)

        # make second violin voice and staff
        second_violin_voice = voicetools.Voice(name='Second Violin Voice')
        second_violin_staff = stafftools.Staff([second_violin_voice], name='Second Violin Staff')
        contexttools.ClefMark('treble')(second_violin_staff)
        instrumenttools.Violin(
            instrument_name_markup='Violin II',
            short_instrument_name_markup='Vl. II'
            )(second_violin_staff)

        # make viola voice and staff
        viola_voice = voicetools.Voice(name='Viola Voice')
        viola_staff = stafftools.Staff([viola_voice], name='Viola Staff')
        contexttools.ClefMark('alto')(viola_staff)
        instrumenttools.Viola()(viola_staff)

        # make cello voice and staff
        cello_voice = voicetools.Voice(name='Cello Voice')
        cello_staff = stafftools.Staff([cello_voice], name='Cello Staff')
        contexttools.ClefMark('bass')(cello_staff)
        instrumenttools.Cello(
            short_instrument_name_markup='Vc.'
            )(cello_staff)

        # make bass voice and staff
        bass_voice = voicetools.Voice(name='Bass Voice')
        bass_staff = stafftools.Staff([bass_voice], name='Bass Staff')
        contexttools.ClefMark('bass')(bass_staff)
        instrumenttools.Contrabass(
            short_instrument_name_markup='Cb.'
            )(bass_staff)

        # make strings staff group
        strings_staff_group = scoretools.StaffGroup([
            first_violin_staff,
            second_violin_staff,
            viola_staff,
            cello_staff,
            bass_staff,
            ],
            name='Strings Staff Group',
            )

        # make score
        score = scoretools.Score([
            bell_staff,
            strings_staff_group,
            ],
            name='Pärt Cantus Score'
            )

        # return Pärt Cantus score
        return score
```

8.2 The bell music

```
def add_bell_music_to_score(score):

    bell_voice = score['Bell Voice']

    def make_bell_phrase():
        phrase = []
        for _ in range(3):
            phrase.append(measuretools.Measure((6, 4), r"r2. a'2. \laissezVibrer"))
            phrase.append(measuretools.Measure((6, 4), 'R1.'))
        for _ in range(2):
            phrase.append(measuretools.Measure((6, 4), 'R1.'))
        return phrase

    for _ in range(11):
        bell_voice.extend(make_bell_phrase())

    for _ in range(19):
        bell_voice.append(measuretools.Measure((6, 4), 'R1.'))

    bell_voice.append(measuretools.Measure((6,4), r"a'1. \laissezVibrer"))
```

8.3 The string music

Creating the music for the strings is a bit more involved, but conceptually falls into two steps. First, we'll procedurally generate basic pitches and rhythms for all string voices. Then, we'll make edits to the generated material by hand. The entire process is encapsulated in the following function:

```
def add_string_music_to_score(score):

    # generate some pitch and rhythm information
    pitch_contour_reservoir = create_pitch_contour_reservoir()
    shadowed_contour_reservoir = shadow_pitch_contour_reservoir(
        pitch_contour_reservoir)
    durated_reservoir = durate_pitch_contour_reservoir(
        shadowed_contour_reservoir)

    # add six dotted-whole notes and the durated contours to each string voice
    for instrument_name, descents in durated_reservoir.iteritems():
        instrument_voice = score['%s Voice' % instrument_name]
        instrument_voice.extend("R1. R1. R1. R1. R1. R1.")
        for descent in descents:
            instrument_voice.extend(descent)

    # apply instrument-specific edits
    edit_first_violin_voice(score, durated_reservoir)
    edit_second_violin_voice(score, durated_reservoir)
    edit_viola_voice(score, durated_reservoir)
    edit_cello_voice(score, durated_reservoir)
    edit_bass_voice(score, durated_reservoir)

    # chop all string parts into 6/4 measures
    for voice in iterationtools.iterate_voices_in_expr(score['Strings Staff Group']):
        for shard in componenttools.split_components_at_offsets(voice[:],
            [(6, 4)], cyclic=True):
            measuretools.Measure((6, 4), shard)
```

The pitch material is the same for all of the strings: a descending a-minor scale, generally decorated with diads. But, each instrument uses a different overall range, with the lower instrument playing slower and slower than the higher instruments, creating a sort of mensuration canon.

For each instrument, the descending scale is fragmented into what we'll call "descents". The first descent uses only the first note of that instrument's scale, while the second descent adds the second note, and the third another. We'll generate as many descents per instruments as there are pitches in its overall scale:

```
def create_pitch_contour_reservoir():

    scale = tonalitytools.Scale('a', 'minor')
    pitch_ranges = {
        'First Violin': pitchtools.PitchRange(("c'", "a'''")),
        'Second Violin': pitchtools.PitchRange(('a', "a''")),
        'Viola': pitchtools.PitchRange(('e', "a'")),
        'Cello': pitchtools.PitchRange(('a,', 'a')),
        'Bass': pitchtools.PitchRange(('c', 'a')),
    }

    reservoir = {}
    for instrument_name, pitch_range in pitch_ranges.iteritems():
        pitch_set = scale.create_named_chromatic_pitch_set_in_pitch_range(pitch_range)
        pitches = sorted(pitch_set.named_chromatic_pitches, reverse=True)
        pitch_descents = []
        for i in xrange(len(pitches)):
            descent = tuple(pitches[:i + 1])
            pitch_descents.append(descent)
        reservoir[instrument_name] = tuple(pitch_descents)

    return reservoir
```

Here's what the first 10 descents for the first violin look like:

```
>>> reservoir = create_pitch_contour_reservoir()
>>> for i in range(10):
...     descent = reservoir['First Violin'][i]
...     print ' '.join(str(x) for x in descent)
...
a'''
a''' g'''
a''' g''' f'''
a''' g''' f''' e'''
a''' g''' f''' e''' d'''
a''' g''' f''' e''' d''' c'''
a''' g''' f''' e''' d''' c''' b''
a''' g''' f''' e''' d''' c''' b'' a''
a''' g''' f''' e''' d''' c''' b'' a'' g''
a''' g''' f''' e''' d''' c''' b'' a'' g'' f''
```

Next we add diads to all of the descents, except for the viola's. We'll use a dictionary as a lookup table, to tell us what interval to add below a given pitch class:

```
def shadow_pitch_contour_reservoir(pitch_contour_reservoir):

    shadow_pitch_lookup = {
        pitchtools.NamedDiatonicPitchClass('a'): -5, # add a P4 below
        pitchtools.NamedDiatonicPitchClass('g'): -3, # add a m3 below
        pitchtools.NamedDiatonicPitchClass('f'): -1, # add a m2 below
        pitchtools.NamedDiatonicPitchClass('e'): -4, # add a M3 below
        pitchtools.NamedDiatonicPitchClass('d'): -2, # add a M2 below
        pitchtools.NamedDiatonicPitchClass('c'): -3, # add a m3 below
        pitchtools.NamedDiatonicPitchClass('b'): -2, # add a M2 below
    }

    shadowed_reservoir = {}

    for instrument_name, pitch_contours in pitch_contour_reservoir.iteritems():
        # The viola does not receive any diads
        if instrument_name == 'Viola':
            shadowed_reservoir['Viola'] = pitch_contours
            continue

        shadowed_pitch_contours = []

        for pitch_contour in pitch_contours[:-1]:
            shadowed_pitch_contour = []
            for pitch in pitch_contour:
                pitch_class = pitch.named_diatonic_pitch_class
                shadow_pitch = pitch + shadow_pitch_lookup[pitch_class]
                diad = (shadow_pitch, pitch)
```

```
                shadowed_pitch_contour.append(diad)
            shadowed_pitch_contours.append(tuple(shadowed_pitch_contour))

        # treat the final contour differently: the last note does not become a diad
        final_shadowed_pitch_contour = []
        for pitch in pitch_contours[-1][:-1]:
            pitch_class = pitch.named_diatonic_pitch_class
            shadow_pitch = pitch + shadow_pitch_lookup[pitch_class]
            diad = (shadow_pitch, pitch)
            final_shadowed_pitch_contour.append(diad)
        final_shadowed_pitch_contour.append(pitch_contours[-1][-1])
        shadowed_pitch_contours.append(tuple(final_shadowed_pitch_contour))

        shadowed_reservoir[instrument_name] = tuple(shadowed_pitch_contours)

    return shadowed_reservoir
```

Finally, we'll add rhythms to the pitch contours we've been constructing. Each string instrument plays twice as slow as the string instrument above it in the score. Additionally, all the strings start with some rests, and use a "long-short" pattern for their rhythms:

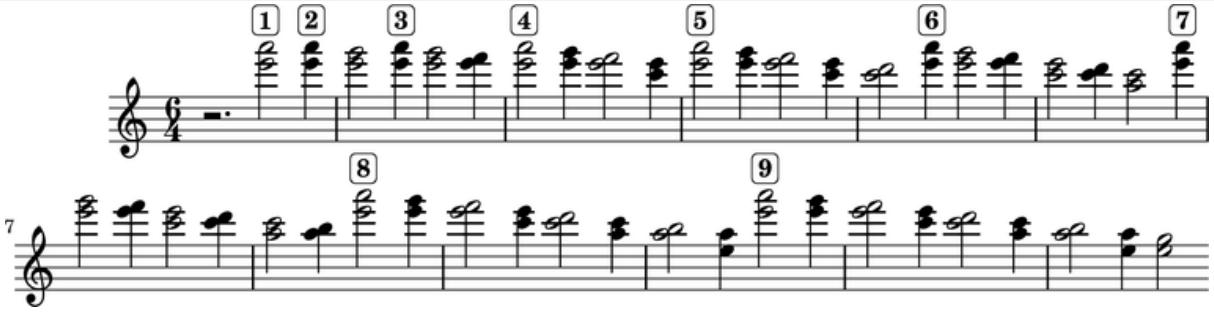
```
def durate_pitch_contour_reservoir(pitch_contour_reservoir):

    instrument_names = [
        'First Violin',
        'Second Violin',
        'Viola',
        'Cello',
        'Bass',
        ]

    durated_reservoir = {}

    for i, instrument_name in enumerate(instrument_names):
        long_duration = Duration(1, 2) * pow(2, i)
        short_duration = long_duration / 2
        rest_duration = long_duration * Multiplier(3, 2)

        div = rest_duration // Duration(3, 2)
        mod = rest_duration % Duration(3, 2)

        initial_rest = resttools.MultiMeasureRest((3, 2)) * div
        if mod:
            initial_rest += resttools.make_rests(mod)

        durated_contours = [tuple(initial_rest)]

        pitch_contours = pitch_contour_reservoir[instrument_name]
        durations = [long_duration, short_duration]
        counter = 0
        for pitch_contour in pitch_contours:
            contour = []
            for pitch in pitch_contour:
                contour.extend(leaftools.make_leaves([pitch], [durations[counter]]))
                counter = (counter + 1) % 2
            durated_contours.append(tuple(contour))

        durated_reservoir[instrument_name] = tuple(durated_contours)

    return durated_reservoir
```

Let's see what a few of those look like. First, we'll build the entire reservoir from scratch, so you can see the process:

```
>>> pitch_contour_reservoir = create_pitch_contour_reservoir()
>>> shadowed_contour_reservoir = shadow_pitch_contour_reservoir(pitch_contour_reservoir)
>>> durated_reservoir = durate_pitch_contour_reservoir(shadowed_contour_reservoir)
```

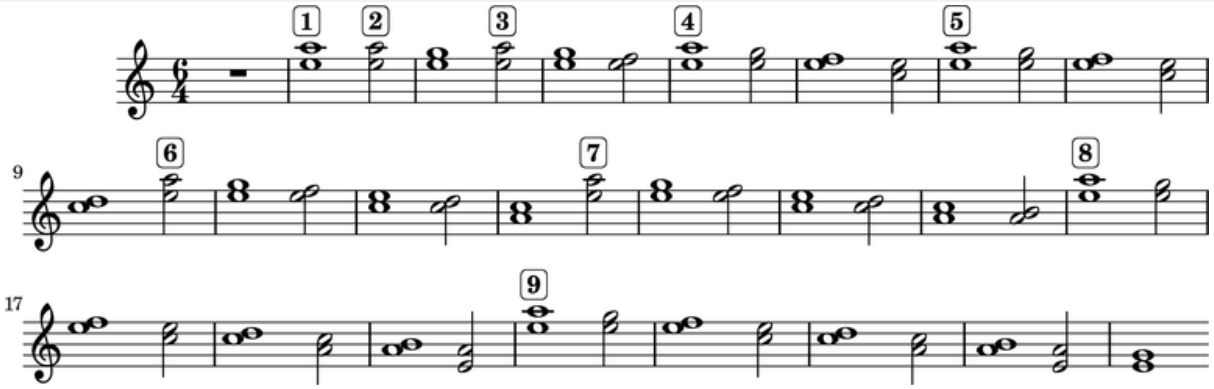
Then we'll grab the sub-reservoir for the first violins, taking the first ten descents (which includes the silences we've been adding as well). We'll label each descent with some markup, to distinguish them, throw them into a Staff and give them a 6/4 time signature, just so they line up properly.

```
>>> descents = durated_reservoir['First Violin'][:10]
>>> for i, descent in enumerate(descents[1:], 1):
...     markup = markuptools.Markup(r'\rounded-box \bold {}'.format(i), Up)(descent[0])
...
>>> staff = Staff(sequencetools.flatten_sequence(descents))
>>> time_signature = contexttools.TimeSignatureMark((6, 4))(staff)
>>> show(staff)
```



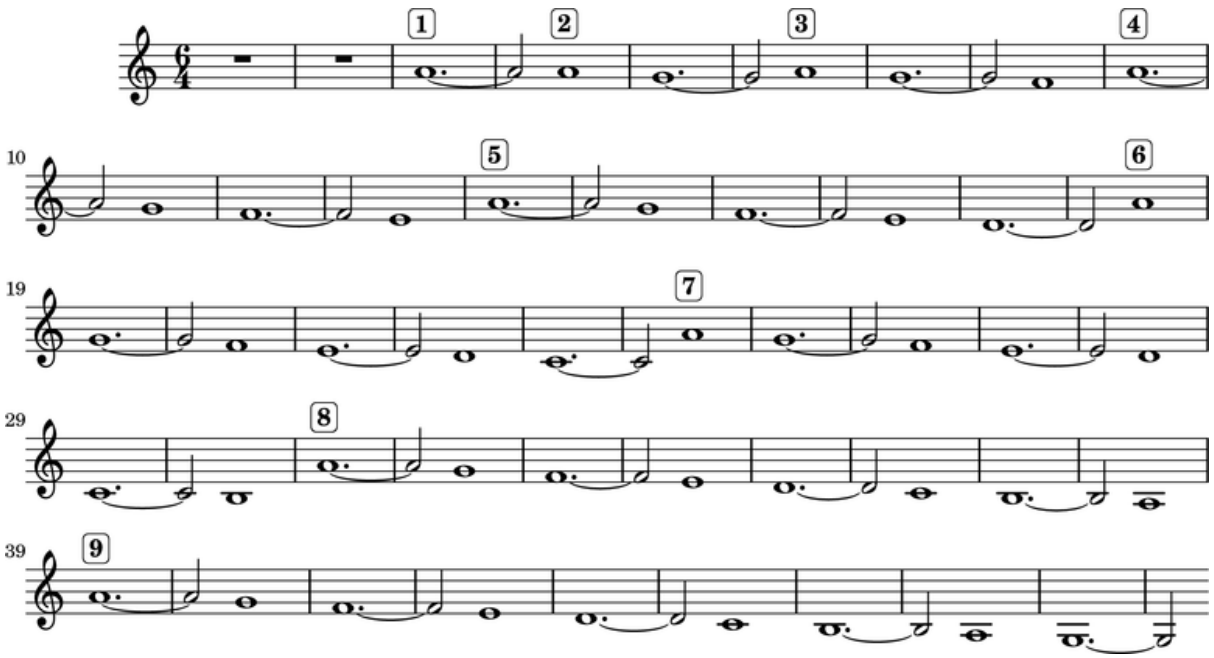
Let's look at the second violins too:

```
>>> descents = durated_reservoir['Second Violin'][:10]
>>> for i, descent in enumerate(descents[1:], 1):
...     markup = markuptools.Markup(r'\rounded-box \bold {}'.format(i), Up)(descent[0])
...
>>> staff = Staff(sequencetools.flatten_sequence(descents))
>>> time_signature = contexttools.TimeSignatureMark((6, 4))(staff)
>>> show(staff)
```



And, last we'll take a peek at the violas. They have some longer notes, so we'll split their music cyclically every 3 half notes, just so nothing crosses the bar lines accidentally:

```
>>> descents = durated_reservoir['Viola'][:10]
>>> for i, descent in enumerate(descents[1:], 1):
...     markup = markuptools.Markup(r'\rounded-box \bold {}'.format(i), Up)(descent[0])
...
>>> staff = Staff(sequencetools.flatten_sequence(descents))
>>> shards = componenttools.split_components_at_offsets(staff[:], [(3, 2)], cyclic=True)
>>> time_signature = contexttools.TimeSignatureMark((6, 4))(staff)
>>> show(staff)
```



You can see how each part is twice as slow as the previous, and starts a little bit later too.

8.4 The edits

```
def edit_first_violin_voice(score, durated_reservoir):

    voice = score['First Violin Voice']
    descents = durated_reservoir['First Violin']

    copied_descent = copy.deepcopy(descents[-1])
    voice.extend(copied_descent)

    final_sustain_rhythm = [(6, 4)] * 43 + [(1, 2)]
    final_sustain_notes = notetools.make_notes(["c'"], final_sustain_rhythm)
    voice.extend(final_sustain_notes)
    tietools.TieSpanner(final_sustain_notes)
    voice.extend('r4 r2.')
```

```
def edit_second_violin_voice(score, durated_reservoir):

    voice = score['Second Violin Voice']
    descents = durated_reservoir['Second Violin']

    copied_descent = list(copy.deepcopy(descents[-1]))
    copied_descent[-1].written_duration = durationtools.Duration(1, 1)
    copied_descent.append(notetools.Note('a2'))
    for leaf in copied_descent:
        marktools.Articulation('accent')(leaf)
        marktools.Articulation('tenuto')(leaf)
    voice.extend(copied_descent)

    final_sustain = []
    for _ in range(32):
        final_sustain.append(notetools.Note('a1.'))
    final_sustain.append(notetools.Note('a2'))
    marktools.Articulation('accent')(final_sustain[0])
    marktools.Articulation('tenuto')(final_sustain[0])

    voice.extend(final_sustain)
    tietools.TieSpanner(final_sustain)
    voice.extend('r4 r2.')
```

```
def edit_viola_voice(score, durated_reservoir):

    voice = score['Viola Voice']
    descents = durated_reservoir['Viola']

    for leaf in descents[-1]:
        marktools.Articulation('accent')(leaf)
        marktools.Articulation('tenuto')(leaf)
    copied_descent = copy.deepcopy(descents[-1])
    for leaf in copied_descent:
        if leaf.written_duration == durationtools.Duration(4, 4):
            leaf.written_duration = durationtools.Duration(8, 4)
        else:
            leaf.written_duration = durationtools.Duration(4, 4)
    voice.extend(copied_descent)

    bridge = notetools.Note('e1')
    marktools.Articulation('tenuto')(bridge)
    marktools.Articulation('accent')(bridge)
    voice.append(bridge)

    final_sustain_rhythm = [(6, 4)] * 21 + [(1, 2)]
    final_sustain_notes = notetools.make_notes(['e'], final_sustain_rhythm)
    marktools.Articulation('accent')(final_sustain_notes[0])
    marktools.Articulation('tenuto')(final_sustain_notes[0])
    voice.extend(final_sustain_notes)
    tietools.TieSpanner(final_sustain_notes)
    voice.extend('r4 r2.')
```

```
def edit_cello_voice(score, durated_reservoir):

    voice = score['Cello Voice']
    descents = durated_reservoir['Cello']

    tie_chain = tietools.get_tie_chain(voice[-1])
    for leaf in tie_chain.leaves:
        parent = leaf.parent
        index = parent.index(leaf)
        parent[index] = chordtools.Chord(['e,', 'a,'], leaf.written_duration)

    unison_descent = copy.deepcopy(voice[-len(descents[-1]):])
    voice.extend(unison_descent)
    for chord in unison_descent:
        index = chord.parent.index(chord)
        parent[index] = notetools.Note(chord.written_pitches[1], chord.written_duration)
        marktools.Articulation('accent')(parent[index])
        marktools.Articulation('tenuto')(parent[index])

    voice.extend('a,1. ~ a,2 b,1 ~ b,1. ~ b,1. a,1. ~ a,1. ~ a,1. ~ a,1. ~ a,1. ~ a,2 r4 r2.')
```

```
def edit_bass_voice(score, durated_reservoir):

    voice = score['Bass Voice']

    voice[-3:] = '<e, e>\maxima <d, d>\longa <c, c>\maxima <b,>\longa <a,>\maxima r4 r2.'
```

8.5 The marks

Now we'll apply various kinds of marks, including dynamics, articulations, bowing indications, expressive instructures, page breaks and rehearsal marks.

We'll start with the bowing marks. This involves creating a piece of custom markup to indicate rebowing. We accomplish this by aggregating together some *markuptools.MarkupCommand* and *markuptools.MusicGlyph* objects. The completed *markuptools.Markup* object is then copied and attached at the correct locations in the score.

Why copy it? A *Mark* can only be attached to a single *Component*. If we attached the original piece of markup to each of our target components in turn, only the last would actually receive the markup, as it would have be detached from the preceding components.

Let's take a look:

```
def apply_bowing_marks(score):

    # apply alternating upbow and downbow for first two sounding bars
    # of the first violin
    for measure in score['First Violin Voice'][6:8]:
        for i, chord in enumerate(iterationtools.iterate_chords_in_expr(measure)):
            if i % 2 == 0:
                marktools.Articulation('downbow')(chord)
            else:
                marktools.Articulation('upbow')(chord)

    # create and apply rebowing markup
    rebow_markup = markuptools.Markup(
        markuptools.MarkupCommand(
            'concat', [
                markuptools.MusicGlyph('scripts.downbow'),
                markuptools.MarkupCommand('hspace', 1),
                markuptools.MusicGlyph('scripts.upbow'),
            ]))
    copy.copy(rebow_markup)(score['First Violin Voice'][64][0])
    copy.copy(rebow_markup)(score['Second Violin Voice'][75][0])
    copy.copy(rebow_markup)(score['Viola Voice'][86][0])
```

After dealing with custom markup, applying dynamics is easy. Just instantiate and attach:

```
def apply_dynamic_marks(score):

    voice = score['Bell Voice']
    contexttools.DynamicMark('ppp')(voice[0][1])
    contexttools.DynamicMark('pp')(voice[8][1])
    contexttools.DynamicMark('p')(voice[18][1])
    contexttools.DynamicMark('mp')(voice[26][1])
    contexttools.DynamicMark('mf')(voice[34][1])
    contexttools.DynamicMark('f')(voice[42][1])
    contexttools.DynamicMark('ff')(voice[52][1])
    contexttools.DynamicMark('fff')(voice[60][1])
    contexttools.DynamicMark('ff')(voice[68][1])
    contexttools.DynamicMark('f')(voice[76][1])
    contexttools.DynamicMark('mf')(voice[84][1])
    contexttools.DynamicMark('pp')(voice[-1][0])

    voice = score['First Violin Voice']
    contexttools.DynamicMark('ppp')(voice[6][1])
    contexttools.DynamicMark('pp')(voice[15][0])
    contexttools.DynamicMark('p')(voice[22][3])
    contexttools.DynamicMark('mp')(voice[31][0])
    contexttools.DynamicMark('mf')(voice[38][3])
    contexttools.DynamicMark('f')(voice[47][0])
    contexttools.DynamicMark('ff')(voice[55][2])
    contexttools.DynamicMark('fff')(voice[62][2])

    voice = score['Second Violin Voice']
    contexttools.DynamicMark('pp')(voice[7][0])
    contexttools.DynamicMark('p')(voice[12][0])
    contexttools.DynamicMark('p')(voice[16][0])
    contexttools.DynamicMark('mp')(voice[25][1])
    contexttools.DynamicMark('mf')(voice[34][1])
    contexttools.DynamicMark('f')(voice[44][1])
    contexttools.DynamicMark('ff')(voice[54][0])
    contexttools.DynamicMark('fff')(voice[62][1])

    voice = score['Viola Voice']
    contexttools.DynamicMark('p')(voice[8][0])
    contexttools.DynamicMark('mp')(voice[19][1])
    contexttools.DynamicMark('mf')(voice[30][0])
    contexttools.DynamicMark('f')(voice[36][0])
    contexttools.DynamicMark('f')(voice[42][0])
    contexttools.DynamicMark('ff')(voice[52][0])
    contexttools.DynamicMark('fff')(voice[62][0])

    voice = score['Cello Voice']
```

```
    contexttools.DynamicMark('p')(voice[10][0])
    contexttools.DynamicMark('mp')(voice[21][0])
    contexttools.DynamicMark('mf')(voice[31][0])
    contexttools.DynamicMark('f')(voice[43][0])
    contexttools.DynamicMark('ff')(voice[52][1])
    contexttools.DynamicMark('fff')(voice[62][0])

    voice = score['Bass Voice']
    contexttools.DynamicMark('mp')(voice[14][0])
    contexttools.DynamicMark('mf')(voice[27][0])
    contexttools.DynamicMark('f')(voice[39][0])
    contexttools.DynamicMark('ff')(voice[51][0])
    contexttools.DynamicMark('fff')(voice[62][0])
```

We apply expressive marks the same way we applied our dynamics:

```
def apply_expressive_marks(score):

    voice = score['First Violin Voice']
    markuptools.Markup(r'\left-column { div. \line { con sord. } }', Up)(voice[6][1])
    markuptools.Markup('sim.', Up)(voice[8][0])
    markuptools.Markup('uniti', Up)(voice[58][3])
    markuptools.Markup('div.', Up)(voice[59][0])
    markuptools.Markup('uniti', Up)(voice[63][3])

    voice = score['Second Violin Voice']
    markuptools.Markup('div.', Up)(voice[7][0])
    markuptools.Markup('uniti', Up)(voice[66][1])
    markuptools.Markup('div.', Up)(voice[67][0])
    markuptools.Markup('uniti', Up)(voice[74][0])

    voice = score['Viola Voice']
    markuptools.Markup('sole', Up)(voice[8][0])

    voice = score['Cello Voice']
    markuptools.Markup('div.', Up)(voice[10][0])
    markuptools.Markup('uniti', Up)(voice[74][0])
    markuptools.Markup('uniti', Up)(voice[84][1])
    markuptools.Markup(r'\italic { espr. }', Down)(voice[86][0])
    markuptools.Markup(r'\italic { molto espr. }', Down)(voice[88][1])

    voice = score['Bass Voice']
    markuptools.Markup('div.', Up)(voice[14][0])
    markuptools.Markup(r'\italic { espr. }', Down)(voice[86][0])
    componenttools.split_components_at_offsets(voice[88][:], [Duration(1, 1), Duration(1, 2)])
    markuptools.Markup(r'\italic { molto espr. }', Down)(voice[88][1])
    markuptools.Markup('uniti', Up)(voice[99][1])

    for voice in iterationtools.iterate_voices_in_expr(score['Strings Staff Group']):
        markuptools.Markup(r'\italic { (non dim.) }', Down)(voice[102][0])
```

We use the *marktools.LilyPondCommandClass* to create LilyPond system breaks, and attach them to measures in the percussion part. After this, our score will break in the exact same places as the original:

```
def apply_page_breaks(score):

    bell_voice = score['Bell Voice']

    measure_indices = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 72,
        79, 86, 93, 100]

    for measure_index in measure_indices:
        marktools.LilyPondCommandMark(
            'break',
            'after'
            )(bell_voice[measure_index])
```

We'll make the rehearsal marks the exact same way we made our line breaks:

```
def apply_rehearsal_marks(score):

    bell_voice = score['Bell Voice']
```

```
    measure_indices = [6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84,
        90, 96, 102]

    for measure_index in measure_indices:
        marktools.LilyPondCommandMark(
            r'mark \default',
            'before'
            )(bell_voice[measure_index])
```

And then we add our final bar lines. *marktools.BarLine* objects inherit from *marktools.Mark*, so you can probably guess by now how we add them to the score... instantiate and attach:

```
def apply_final_bar_lines(score):

    for voice in iterationtools.iterate_voices_in_expr(score):
        marktools.BarLine('|.')(voice[-1])
```

8.6 The LilyPond file

Finally, we create some functions to apply formatting directives to our *Score* object, then wrap it into a *LilyPondFile* and apply some more formatting.

In our *configure_score()* functions, we use *layouttools.make_spacing_vector()* to create the correct Scheme construct to tell LilyPond how to handle vertical space for its staves and staff groups. You should consult LilyPond's vertical spacing documentation for a complete explanation of what this Scheme code means:

```
>>> spacing_vector = layouttools.make_spacing_vector(0, 0, 8, 0)
>>> f(spacing_vector)
#'((basic_distance . 0) (minimum_distance . 0) (padding . 8) (stretchability . 0))
```

```
def configure_score(score):

    spacing_vector = layouttools.make_spacing_vector(0, 0, 8, 0)
    score.override.vertical_axis_group.staff_staff_spacing = spacing_vector
    score.override.staff_grouper.staff_staff_spacing = spacing_vector
    score.override.staff_symbol.thickness = 0.5
    score.set.mark_formatter = schemetools.Scheme('format-mark-box-numbers')
```

In our *configure_lilypond_file()* function, we need to construct a ContextBlock definition in order to tell LilyPond to hide empty staves, and additionally to hide empty staves if they appear in the first system:

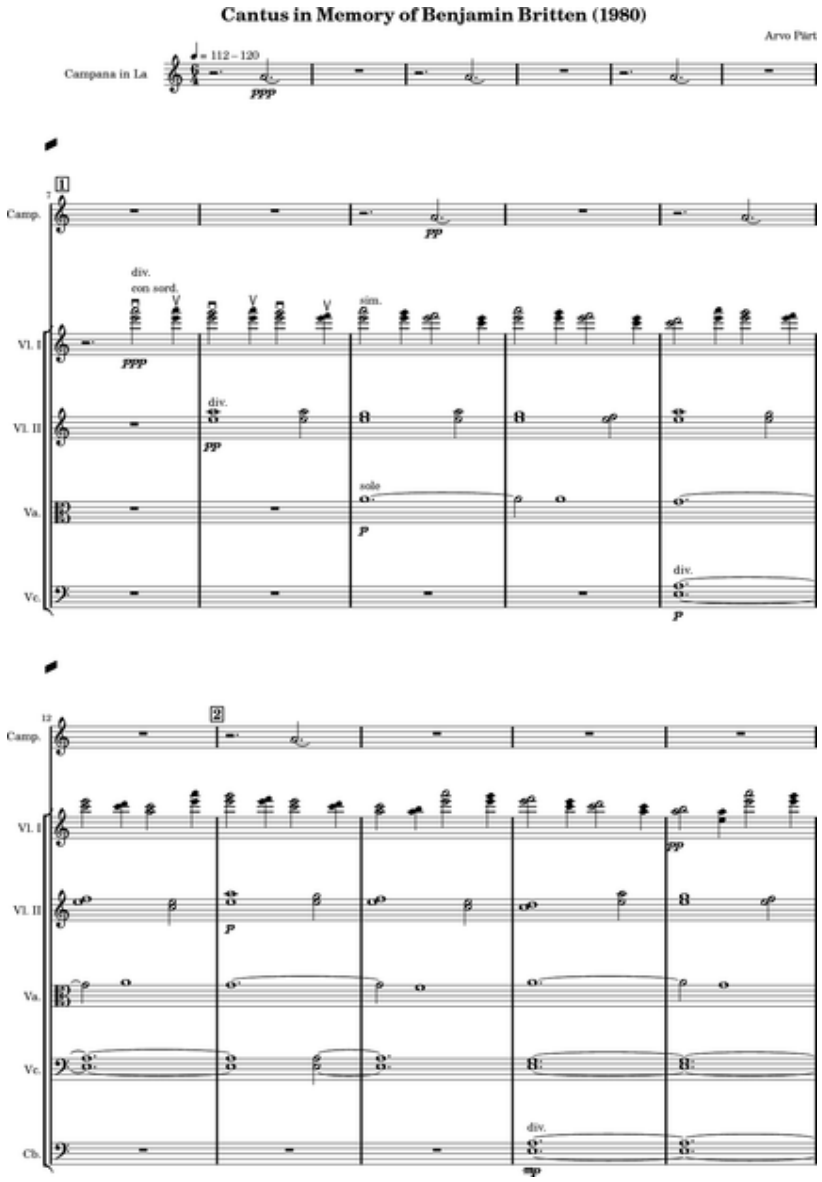
```
def configure_lilypond_file(lilypond_file):

    lilypond_file.global_staff_size = 8

    context_block = lilypondfiletools.ContextBlock()
    context_block.context_name = r'Staff \RemoveEmptyStaves'
    context_block.override.vertical_axis_group.remove_first = True
    lilypond_file.layout_block.context_blocks.append(context_block)

    lilypond_file.paper_block.system_separator_markup = marktools.LilyPondCommandMark('slashSeparator')
    lilypond_file.paper_block.bottom_margin = lilypondfiletools.LilyPondDimension(0.5, 'in')
    lilypond_file.paper_block.top_margin =    lilypondfiletools.LilyPondDimension(0.5, 'in')
    lilypond_file.paper_block.left_margin =   lilypondfiletools.LilyPondDimension(0.75, 'in')
    lilypond_file.paper_block.right_margin =  lilypondfiletools.LilyPondDimension(0.5, 'in')
    lilypond_file.paper_block.paper_width =   lilypondfiletools.LilyPondDimension(5.25, 'in')
    lilypond_file.paper_block.paper_height =  lilypondfiletools.LilyPondDimension(7.25, 'in')

    lilypond_file.header_block.composer = markuptools.Markup('Arvo Pärt')
    lilypond_file.header_block.title = markuptools.Markup('Cantus in Memory of Benjamin Britten (1980)')
```

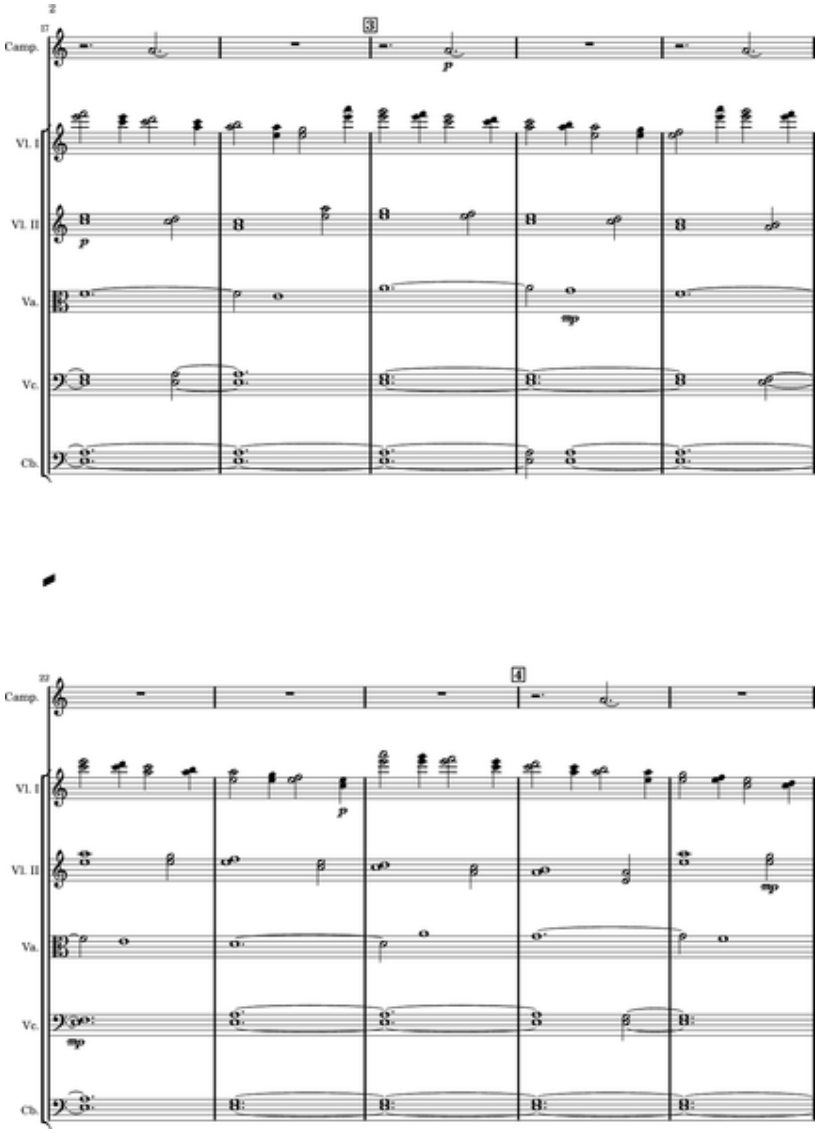
Let's run our original toplevel function to build the complete score:

```
>>> lilypond_file = make_part_lilypond_file()
```

And here we show it:

```
>>> show(lilypond_file)
```




Camp.
Vl. I
Vl. II
Va.
Vc.
Cb.

Part III

System Overview

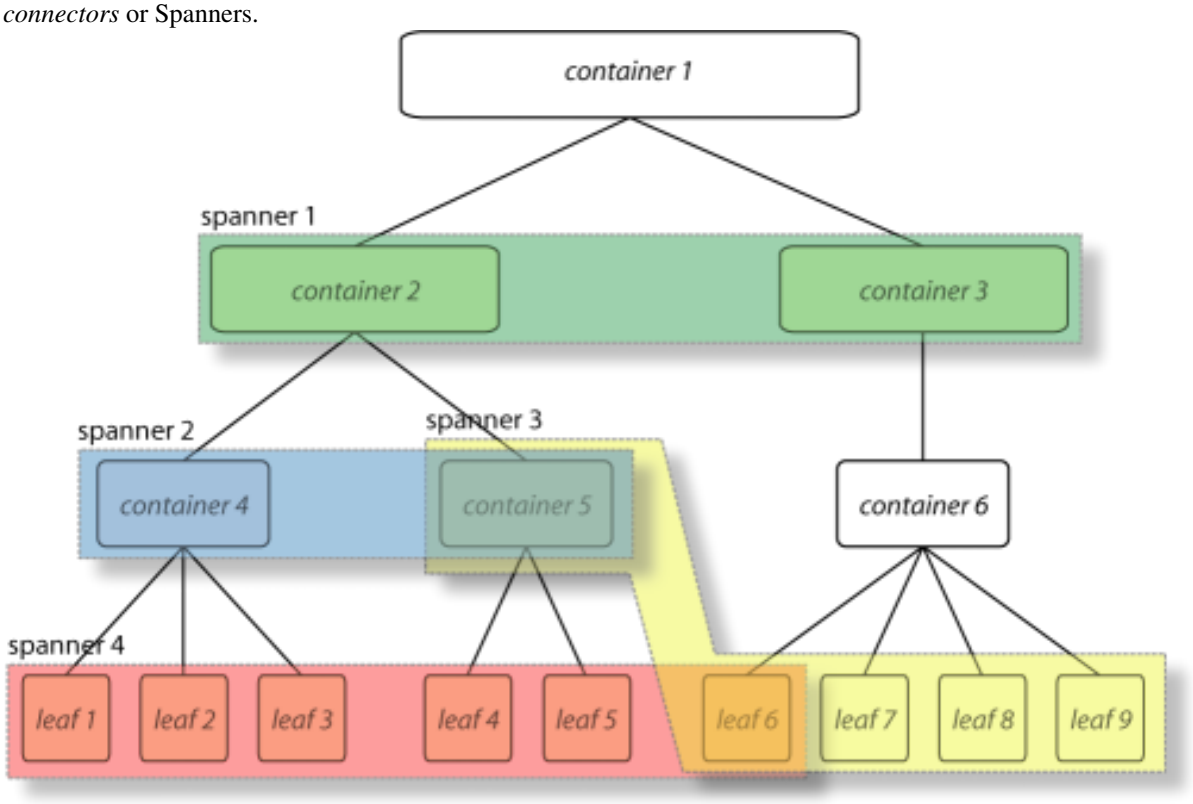
CHAPTER

NINE

LEAF, CONTAINER, SPANNER, MARK

At the heart of Abjad's Symbolic Score-Control lies a powerful model that we call the Leaf Container Spanner Mark, or LCSM, model of the musical score.

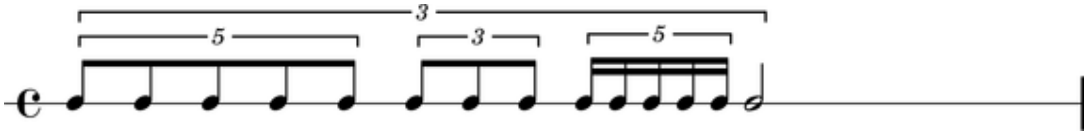
The LCSM model can be schematically visualized as a superposition of two complementary and completely independent layers of structure: a *tree* that includes the Containers and the Leaves, and a layer of free floating *connectors* or Spanners.



There can be any number of Spanners, they may overlap, and they may connect to different levels of the tree hierarchy. The spanner attach to the elements of the tree, so a tree structure must exist for spanners to be made manifest.

9.1 Example 1

To understand the whys and hows of the LCSM model implemented in Abjad, it is probably easier to base the discussion on concrete musical examples. Let's begin with a simple and rather abstract musical fragment: a measure with nested tuplets.



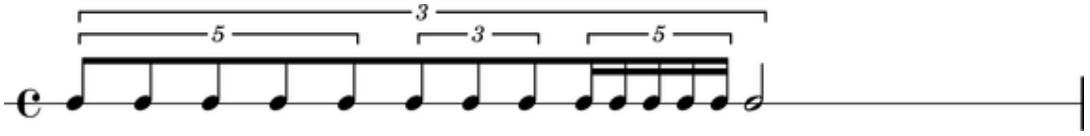
What we see in this little fragment is a measure with 4/4 meter, 14 notes and four tuplet brackets prolating the notes. The three bottom tuplets (with ratios 5:4, 3:2, 5:4) prolate all but the last note. The topmost tuplet prolates all the notes in the measure and combines with the bottom three tuplets to doubly prolate all but the last note. The topmost tuplet as thus prolates three tuplets, each of which in turn prolates a group of notes. We can think of a tuplet as *containing* notes or other tuplets or both. Thus, in our example, the topmost tuplet contains three tuplets and a half note. Each of the tuplets contained by the topmost tuplet in turn contains five, three, and five notes respectively. If we add the measure, then we have a measure that contains a tuplet that contains tuplets that contain notes. The structure of the measure with nested tuplets as we have just described it has two important properties:

1. It is a *hierarchical* structure.
2. It follows *exclusive membership*, meaning that each element in the hierarchy (a note, a tuplet or a measure) has one and only one *parent*. In other words a single note is not contained in more than one tuplet simultaneously, and no one tuplet is contained in more than one other tuplet at the same time.

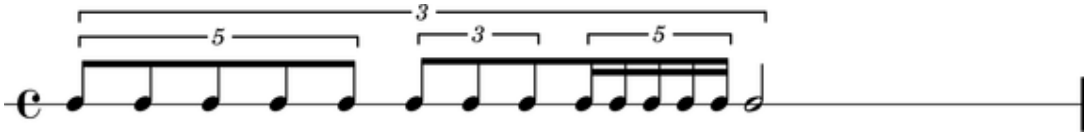
What we are describing here is a tree, and it is the structure of Abjad *containers*.

While this tree structure seem like the right way to represent the relationships between the elements of a score, it is not enough. Consider the tuplet example again with the following beaming alternatives:

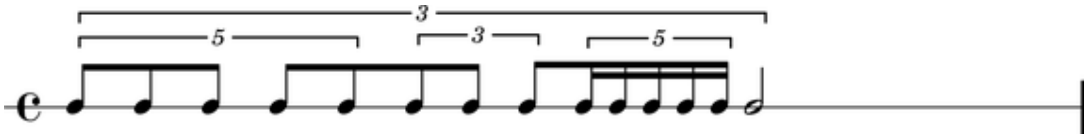
Beaming alternative 1:



Beaming alternative 2:



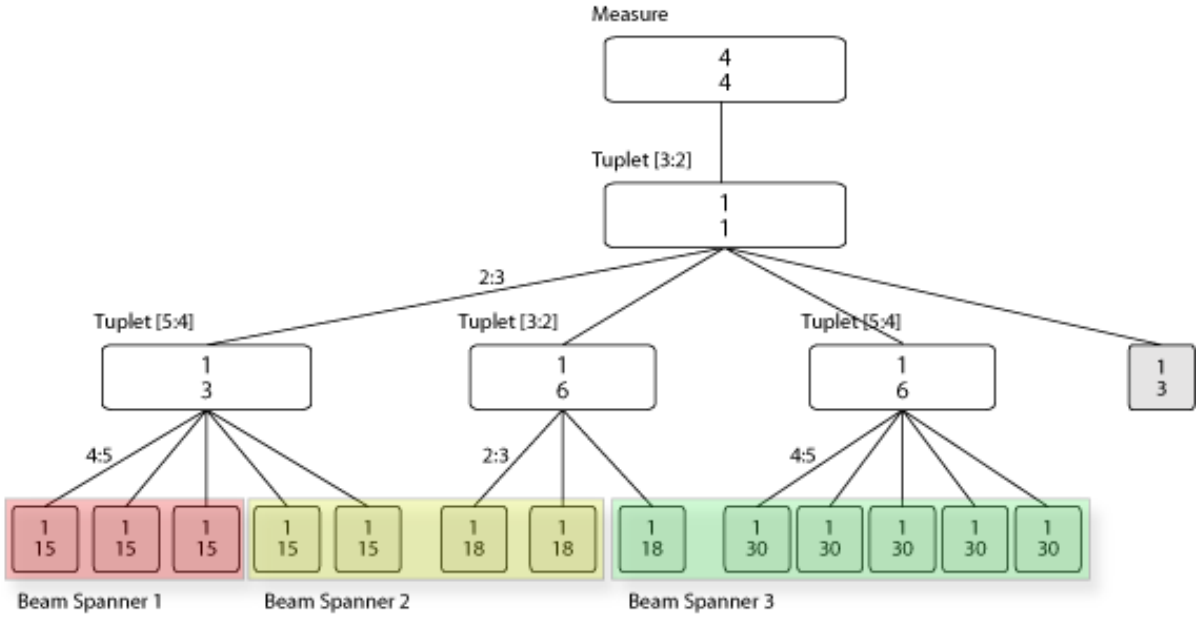
Beaming alternative 3:



Clearly the beaming of notes can be totally independent from the tuplet groupings. Beaming across tuplet groups implies beaming across nodes in the tree structure, which means that the beams do not adhere to the *exclusive (parenthood) membership* characteristic of the tree. Beams must then be modeled independently as a separate and complementary structure. These are the Abjad *spanners*.

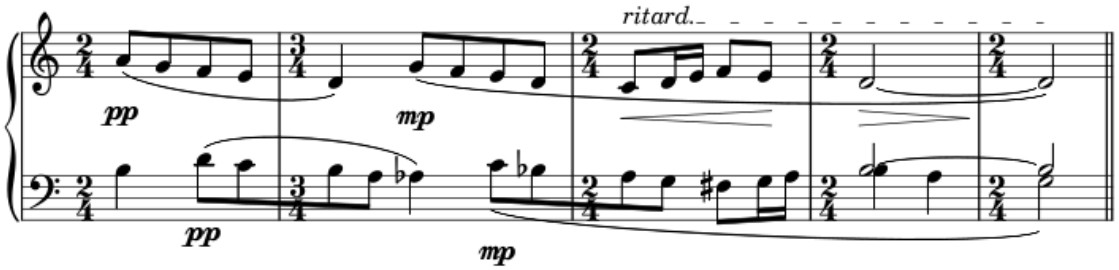
Below we have the score of our tuplet example with alternative beaming and its the Leaf-Container-Spanner graph. Notice that the colored blocks represent spanners.

Beaming alternative 3 (graph):



9.2 Example 2

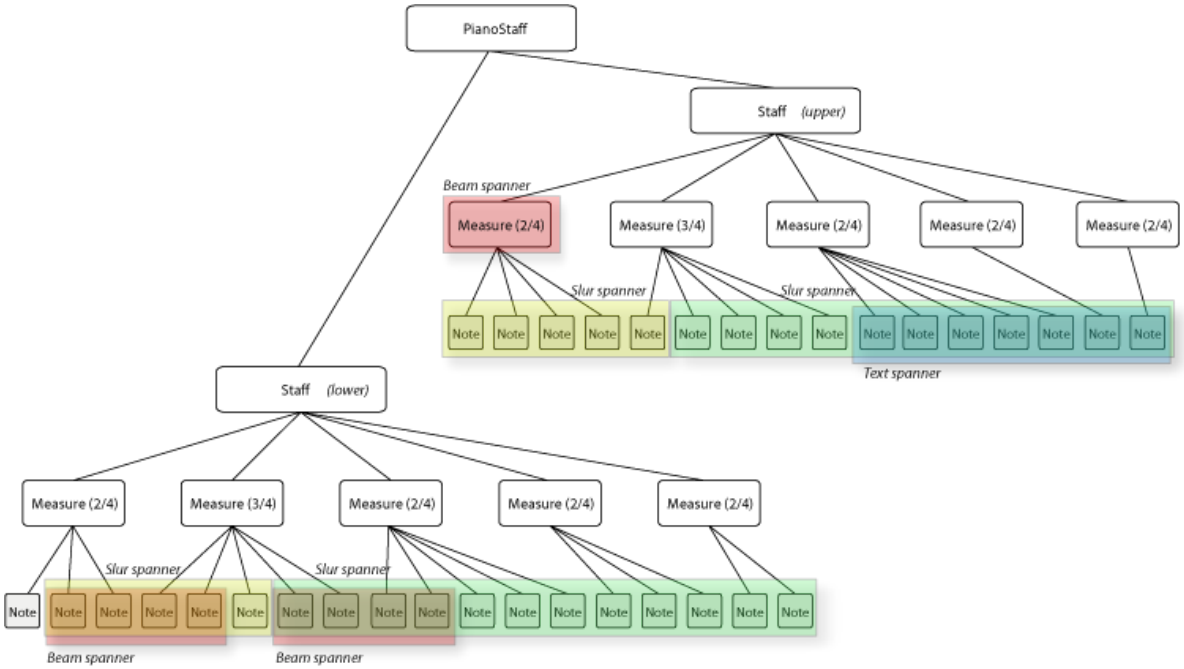
As a second example let's look at the last five measures of Bartók's *Wandering* from Mikrokosmos vol. III. As simple as it may seem, these five measures carry with them a lot of information pertaining to musical notation.



Note: Please refer to the *Bartok example* for a step by step construction of the musical fragment and its full Abjad code.

There are many musical signs of different types on the pages: notes, dynamic markings, clefs, staves, slurs, etc. These signs are structurally related to each other in different ways. Let's start by looking at the larger picture. The piano piece is written in two staves. As is customary, the staves are graphically grouped with a large curly brace attaching to them at the beginning or each system. Notice that each staff has a variety of signs associated with it. There are notes printed on the staff lines as well as meter indications and bar lines. Each note, for example, is in one and only one staff. A note is never in two staves at the same time. This is also true for measures. A measure in the top staff is not simultaneously drawn on the top staff and the bottom staff. It is better to think of each staff as having its own set of measures. Notice also that the notes in each staff fall within the region of one and only one measure, i.e. measures seem to contain notes. There is not one note that is at once in two measures (this is standard practice in musical notation, but it need not always be the case.)

As we continue describing the relationships between the musical signs in the page, we begin to discover a certain structure, or a convenient way of structuring the score for conceptualization and manipulation. All the music in a piano score seems to be written in what we might call a *staff group*. The staff group is *composed of* two staves. Each staff in turn appears to be composed of a series or measures, and each measure is composed of a series of notes. So again we find that the score structure can be organized hierarchically as a tree. This tree structure looks like this:



Notice again though that there are elements in the score that imply and require a different kind of grouping. The two four eighth-note runs in the lower staff are beamed together across the bar line and, based on our tree structure, across tree nodes. So do the slurs, the dynamics markings and the ritardando indication at the top of the score. As we have seen in the tuplets example, all these groups running across the tree structure can be defined with *spanners*.

CHAPTER

TEN

PARSING

Abjad provides a growing number of language parsers. The most important of these is a sophisticate LilyPond parser.

10.1 LilyPond Parsing

`lilypondparsertools.LilyPondParser` parses a large, although incomplete, subset of LilyPond's syntax:

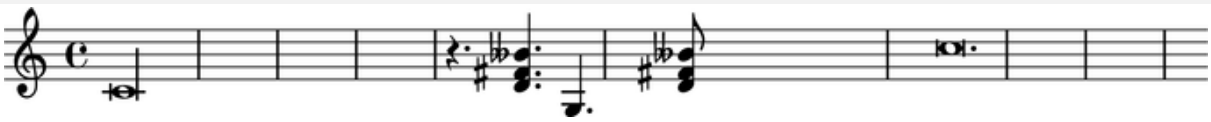
```
>>> parser = lilypondparsertools.LilyPondParser()
```

The LilyPond parser understands notes, chords, skips and rests, including default durations and the `q` chord-repeat construct:

```
>>> string = r"{ c'\longa r4. <d' fs' bff'> g q8 s1 c''\breve. }"
>>> result = parser(string)
```

```
>>> f(result)
{
    c'\longa
    r4.
    <d' fs' bff'>4.
    g4.
    <d' fs' bff'>8
    s1
    c''\breve.
}
```

```
>>> show(result)
```

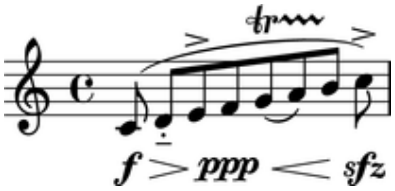


The LilyPond parser understands most spanners, articulations and dynamics too:

```
>>> string = r'''\new Staff {
...     c'8 \f \> (
...     d' -_ [
...     e' ^>
...     f' \ppp \<
...     g' \startTrillSpan \(
...     a' \)
...     b' ] \stopTrillSpan
...     c'' ) \accent \sfz
... }
... '''
>>> result = parser(string)
```

```
>>> f(result)
\new Staff {
    c'8 \f \> (
    d'8 -\portato [
    e'8 ^\accent
    f'8 \ppp \<
    g'8 \( \startTrillSpan
    a'8 \)
    b'8 ] \stopTrillSpan
    c''8 -\accent \sfz )
}
```

```
>>> show(result)
```

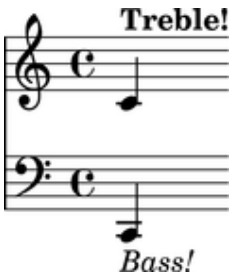


The LilyPond parser understands contexts and markup:

```
>>> string = r'''\new Score <<
...     \new Staff = "Treble Staff" {
...         \new Voice = "Treble Voice" {
...             c' ^\markup { \bold Treble! }
...         }
...     }
...     \new Staff = "Bass Staff" {
...         \new Voice = "Bass Voice" {
...             \clef bass
...             c, _\markup { \italic Bass! }
...         }
...     }
... >>
... '''
>>> result = parser(string)
```

```
>>> f(result)
\new Score <<
    \context Staff = "Treble Staff" {
        \context Voice = "Treble Voice" {
            c'4
                ^ \markup {
                    \bold
                        Treble!
                    }
        }
    }
    \context Staff = "Bass Staff" {
        \context Voice = "Bass Voice" {
            \clef "bass"
            c,4
                _ \markup {
                    \italic
                        Bass!
                    }
        }
    }
>>
```

```
>>> show(result)
```



The LilyPond parser even understands certain aspects of LilyPond file layouts, like header blocks:

```
>>> string = r'''
... \header {
...     name = "Foo von Bar"
...     composer = \markup { by \bold \name }
...     title = \markup { The ballad of \name }
...     tagline = \markup { "" }
... }
... \score {
...     \new Staff {
...         \time 3/4
...         g' ( b' d'' )
...         e''4. ( c''8 c'4 )
...     }
... }
... '''
>>> result = parser(string)
```

```
>>> f(result)
% Abjad revision 8302:8306
% 2012-12-15 16:23

\version "2.16.1"
\language "english"
\include "/media/Work/dev/scores/abjad/trunk/abjad/cfg/abjad.scm"

\header {
    composer = \markup {
        by
        \bold
            "Foo von Bar"
        }
    name = #"Foo von Bar"
    tagline = \markup {  }
    title = \markup {
        The
        ballad
        of
        "Foo von Bar"
        }
}

\score {
    \new Staff {
        \time 3/4
        g'4 (
        b'4
        d''4 )
        e''4. (
        c''8
        c'4 )
    }
}
```

```
>>> show(result)
```

A small number of music functions are also supported, such as `\relative`. Music functions which mutate the score during compilation, result in a normalized Abjad score structure. That is, the resulting Abjad structure corresponds to the music as it appears on the page:

```
>>> string = r'''\new Staff \relative c { c32 d e f g a b c d e f g a b c d e f g a b c }'''
>>> result = parser(string)
```

```
>>> f(result)
\new Staff {
    c32
    d32
    e32
    f32
    g32
    a32
    b32
    c'32
    d'32
    e'32
    f'32
    g'32
    a'32
    b'32
    c''32
    d''32
    e''32
    f''32
    g''32
    a''32
    b''32
    c'''32
}
```

```
>>> show(result)
```

10.2 RhythmTree Parsing

`rhythmtreetools.RhythmTreeParser` parses a microlanguage resembling Ircam's RTM-style LISP syntax, and generates a sequence of RhythmTree structures, which can be furthered manipulated by composers, before being converted into Abjad score object:

```
>>> parser = rhythmtreetools.RhythmTreeParser()
```

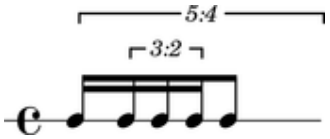
```
>>> string = '(1 (1 (2 (1 1 1)) 2))'
>>> result = parser(string)
>>> result[0]
RhythmTreeContainer(
    children=(
        RhythmTreeLeaf(
            duration=durationtools.Duration(1, 1),
            is_pitched=True,
            ),
        RhythmTreeContainer(
```

```
            children=(
                RhythmTreeLeaf(
                    duration=durationtools.Duration(1, 1),
                    is_pitched=True,
                    ),
                RhythmTreeLeaf(
                    duration=durationtools.Duration(1, 1),
                    is_pitched=True,
                    ),
                RhythmTreeLeaf(
                    duration=durationtools.Duration(1, 1),
                    is_pitched=True,
                    ),
            ),
            duration=Duration(2, 1)
            ),
        RhythmTreeLeaf(
            duration=durationtools.Duration(2, 1),
            is_pitched=True,
            ),
    ),
    duration=Duration(1, 1)
    )
```

```
>>> tuplet = result[0]((1, 4))[0]
>>> f(tuplet)
\times 4/5 {
    c'16
    \times 2/3 {
        c'16
        c'16
        c'16
    }
    c'8
}
```

```
>>> staff = stafftools.RhythmicStaff([tuplet])
```

```
>>> show(staff, docs=True)
```



10.3 "Reduced-Ly" Parsing

`lilypondparsertools.ReducedLyParser` parses the "reduced-ly" microlanguage, whose syntax combines a very small subset of LilyPond syntax, along with affordances for generating various types of Abjad containers, and speedups for rapidly notating notes and rests without needing to specify pitches. It used mainly for creating Abjad documentation:

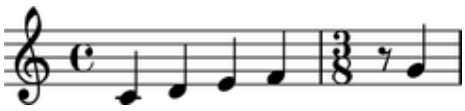
```
>>> parser = lilypondparsertools.ReducedLyParser()
```

```
>>> string = "| 4/4 c' d' e' f' || 3/8 r8 g'4 |"
>>> result = parser(string)
```

```
>>> f(result)
{
    {
        \time 4/4
        c'4
        d'4
        e'4
        f'4
    }
```

```
    {
        \time 3/8
        r8
        g'4
    }
}
```

```
>>> show(result)
```



Part IV

Tutorials

CHAPTER

ELEVEN

GETTING STARTED

Abjad makes powerful programming techniques available to you when you compose. Read through the points below and then click next to proceed.

11.1 Knowing your operating system

Before you start working with Abjad you should review the command line basics of your operating system. You should know how move around the filesystem, how to list the contents of directories and how to copy files. You should know enough about environment variables to make sure that your operating system knows where Abjad is installed. You might also consider installing any OS updates on your computer, too, since you'll need Python 2.7 to run Abjad. When you start building score with Abjad you'll find the system to be almost entirely platform-independent.

11.2 Chosing a text editor

You'll edit many text files when you work with Abjad. So you'll want to spend some time picking out a text editor before you begin. If this is your first time programming you might want to Google and read what other programmers have to say on the matter. Or you could ask a programmer friend about the editor she prefers. Linux programmers sometimes like `vi` or `emacs`. Macintosh programmers might prefer `TextMate`. Whatever your choice make sure you set your editor is set to produce plain text files before you start.

11.3 Launching the terminal

To work with Abjad you'll need a terminal window. The way that you open the terminal window depends on your computer. If you're using MacOS X you can navigate from `Applications` to `Utilities` and then click on `Terminal`. Linux and Windows house the terminal elsewhere. Regardless of the terminal client you chose the purpose of the terminal is to let you type commands to your computer's operating system.

11.4 Where to save your work

Where you choose to save the files you create with Abjad is up to you. Eventually you'll want to create a dedicated set of directories to organize your work. But for now you can create the files described in the tutorials on your desktop, in your documents folder or anywhere else you like.

CHAPTER

TWELVE

LILYPOND "HELLO, WORLD!"

Working with Abjad means working with LilyPond.

To start we'll need to make sure LilyPond is installed.

Open the terminal and type `lilypond --version`:

```
$ lilypond --version
GNU LilyPond 2.17.3

Copyright (c) 1996--2012 by
  Han-Wen Nienhuys <hanwen@xs4all.nl>
  Jan Nieuwenhuizen <janneke@gnu.org>
  and others.

This program is free software.  It is covered by the GNU General Public
License and you are welcome to change it and/or distribute copies of it
under certain conditions.  Invoke as `lilypond --warranty' for more
information.
```

LilyPond responds with version and copyright information. If the terminal tells you that LilyPond is not found then either LilyPond isn't installed on your computer or else your computer doesn't know where LilyPond is installed.

If you haven't installed LilyPond go to `www.lilypond.org` and download the current version of LilyPond for your operating system.

If your computer doesn't know where LilyPond is installed then you'll have to tell your computer where LilyPond is. Doing this depends on your operating system. If you're running MacOS X or Linux then you need to make sure that the location of the LilyPond binary is present in your `PATH` environment variable. If you don't know how to add things to your path you should Google or ask a friend.

12.1 Writing the file

Change to whatever directory you'd like and then use your text editor to create a new file called `hello_world.ly`.

Type the following lines of LilyPond input into `hello_world.ly`:

```
\version "2.17.3"
\language "english"

\score {
    c'4
}
```

Save `hello_world.ly` and quit your text editor when you're done.

Note the following:

```
1. You can use either spaces or tabs while you type.
2. The version string you type must match the LilyPond version you found above.
3. The English language command tells LilyPond to use English note names.
4. The score block tells LilyPond that you're entering actual music.
5. The expression c'4 tells LilyPond to create a quarter note middle C.
6. LilyPond files end in .ly by convention.
```

12.2 Interpreting the file

Call LilyPond on `hello_world.ly`:

```
$ lilypond hello_world.ly
GNU LilyPond 2.17.3
Processing `hello_world.ly'
Parsing...
Interpreting music...
Preprocessing graphical objects...
Finding the ideal number of pages...
Fitting music on 1 page...
Drawing systems...
Layout output to `hello_world.ps'...
Converting to `./hello_world.pdf'...
Success: compilation successfully completed
```

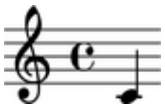
LilyPond reads `hello_world.ly` as input and creates `hello_world.pdf` as output.

Open the `hello_world.pdf` file LilyPond creates.

You can do this by clicking on the file. Or you can open the file from the command line.

If you're using MacOS X you can open `hello_world.pdf` like this:

```
$ open hello_world.pdf
```



Your operating system shows the score you created.

12.3 Repeating the process

Working with LilyPond means doing these things:

```
1. edit a LilyPond input file
2. interpet the input file
3. open the PDF and inspect your work
```

You'll repeat this process many times to make your scores look the way you want. But no matter how complex your music this edit-interpret-view loop will be the basic way you work.

CHAPTER

THIRTEEN

PYTHON "HELLO, WORLD!" (AT THE INTERPRETER)

Working with Abjad means programming in Python. Let's start with Python's interactive interpreter.

13.1 Starting the interpreter

Open the terminal and type `python` to start the interpreter:

```
$ python
```

Python responds with version information and a prompt:

```
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr  9 2012, 20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The purpose of the interpreter is to let you try out code one line at a time.

13.2 Entering commands

Type the following at the interpreter's prompt:

```
>>> print 'hello, world!'
hello, world!
```

Python responds by printing `hello, world!` to the terminal.

13.3 Stopping the interpreter

Type `quit()`. Or type the two-key combination `ctrl+D`:

```
>>> quit()
```

The interpreter stops and returns you to the terminal.

The Python interpreter is a good way to do relatively small things quickly.

But as your projects become more complex you will want to organize the code you write in files.

This is the topic of the next tutorial.

CHAPTER

FOURTEEN

PYTHON "HELLO, WORLD!" (IN A FILE)

This tutorial recaps the Python "hello, world!" of the previous the tutorial. The difference is that here you'll save the code you write to disk.

14.1 Writing the file

Change to whatever directory you'd like and then use your text editor to create a new file called `hello_world.py`.

Type the following line of Python code into `hello_world.py`:

```
print 'hello, world!'
```

Save `hello_world.py` when you're done.

14.2 Interpreting the file

Open the terminal and call Python on `hello_world.py`:

```
$ python hello_world.py
hello, world!
```

Python reads `hello_world.py` as input and outputs `hello, world!` to the terminal.

14.3 Repeating the process

Working with Python files means doing these things:

```
1. write a file
2. interpret the file
3. repeat 1 - 2
```

Experience will make this edit-interpret loop familiar. And no matter how complicated the projects you develop this way of working with Python files will stay the same.

CHAPTER

FIFTEEN

MORE ABOUT PYTHON

The tutorials earlier in this section showed basic ways to work with Python. In this tutorial we'll use the interactive interpreter to find out more about the language and library of tools that it contains.

15.1 Doing many things

You can use the Python interpreter to do many things.

Simple math like addition looks like this:

```
>>> 2 + 2
4
```

Exponentiation looks like this:

```
>>> 2 ** 38
274877906944
```

Interacting with the Python interpreter means typing something as input that Python then evaluates and prints as output.

As you learn more about Abjad you'll work more with Python files than with the Python interpreter. But the Python interpreter's input-output loop makes it easy to see what Python is all about.

15.2 Looking around

Use `dir()` to see the things the Python interpreter knows about:

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

These four things are the only elements that Python loads into the so-called global namespace when you start the interpreter.

Now let's define the variable `x`:

```
>>> x = 10
```

Which lets us do things with `x`:

```
>>> x ** 2
100
```

When we call `dir()` now we see that the global namespace has changed:

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'x']
```

Using `dir()` is a good way to check the variables Python knows about when it runs.

Now type `__builtins__` at the prompt:

```
>>> __builtins__
<module '__builtin__' (built-in)>
```

Python responds and tells us that `__builtins__` is the name of a module.

A module is file full of Python code that somebody has written to provide new functionality.

Use `dir()` to inspect the contents of `__builtins__`:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError', 'BytesWarning',
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'IndentationError',
'IndexError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None',
'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning',
'ReferenceError', 'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning',
'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
'__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring', 'bin', 'bool', 'buffer',
'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',
'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter',
'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long', 'map', 'max',
'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range',
'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

Python responds with a list of many names.

Use Python's `len()` command together with the last-output character `_` to find out how many names `__builtins__` contains:

```
>>> len(_)
144
```

These names make up the core of the Python programming language.

As you learn Abjad you'll use some Python built-ins all the time and others less often.

Before moving on, notice that both `dir()` and `len()` appear in the list above. This explains why we've been able to use these commands in this tutorial.

CHAPTER

SIXTEEN

ABJAD "HELLO, WORLD" (AT THE INTERPRETER)

16.1 Starting the interpreter

Open the terminal and start the Python interpreter:

```
abjad$ python
```

```
Python 2.7.3 (v2.7.3:70274d53c1dd, Apr  9 2012, 20:52:43)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Then import Abjad:

```
>>> from abjad import *
```

If Abjad is installed on your system then Python will silently load Abjad. If Abjad isn't installed on your system then Python will raise an import error.

Go to `www.projectabjad.org` and follow the instructions there to install Abjad if necessary.

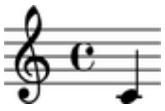
16.2 Entering commands

After you've imported Abjad you can create a note like this:

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

And you can show the note like this:

```
>>> show(note)
```



16.3 Stopping the interpreter

Type `quit()` or `ctrl+D` when you're done:

```
>>> ^D
```

Working with the interpreter is a good way to test out small bits of code in Abjad. As your scores become more complex you will want to organize the code your write with Abjad in files. This is the topic of the next tutorial.

CHAPTER

SEVENTEEN

ABJAD "HELLO, WORLD!" (IN A FILE)

17.1 Writing the file

Open the terminal and change to whatever directory you'd like.

Use your text editor to create a new file called `hello_world.py`. If you have `hello_world.py` left over from earlier you should delete it and create a new file.

Type the following lines of code into `hello_world.py`:

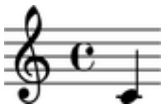
```
from abjad import *


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

Save `hello_world.py` and quit your text editor.

17.2 Interpreting the file

Call Python on `hello_world.py`:

```
$ python hello_world.py
```



Python reads `hello_world.py` and shows the score you've created.

17.3 Repeating the process

Working with files in Abjad means that you do these things:

```
1. edit a file
2. interpret the file
```

These steps make up a type of edit-interpret loop.

This way of working with Abjad remains the same no matter how complex the scores you build.

CHAPTER

EIGHTEEN

MORE ABOUT ABJAD

18.1 How it works

How does Python suddenly know what musical notes are? And how to make musical score?

Use Python's `dir()` built-in to get a sense of the answer:

```
>>> dir()
['ABJCFG', 'Chord', 'Container', 'Duration', 'Fraction', 'Measure', 'Note', 'Rest', 'Score', 'Staff',
'Tuplet', 'Voice', '__builtins__', '__doc__', '__name__', '__package__', '__warningregistry__', 'abctools',
'abjadbooktools', 'beamtools', 'chordtools', 'componenttools', 'configurationtools', 'containertools',
'contexttools', 'datastructuretools', 'decoratortools', 'developerscripttools', 'documentationtools',
'durationtools', 'exceptiontools', 'f', 'formattools', 'gracetools', 'importtools', 'instrumenttools',
'introspectiontools', 'iotools', 'iterationtools', 'labeltools', 'layouttools', 'leaftools',
'lilypondfiletools', 'lilypondparsertools', 'lilypondproxytools', 'marktools', 'markuptools', 'mathtools',
'measuretools', 'notetools', 'offsettools', 'p', 'pitcharraytools', 'pitchtools', 'play', 'resttools',
'rhythmtreetools', 'schemetools', 'scoretemplatetools', 'scoretools', 'sequencetools', 'show', 'sievetools',
'skiptools', 'spannertools', 'stafftools', 'stringtools', 'tempotools', 'tietools', 'timeintervaltools',
'timesignaturetools', 'rhythmmakertools', 'tonalitytools', 'tuplettools', 'verticalitytools', 'voicetools',
'wellformednesstools', 'z']
```

Calling `from abjad import *` causes Python to load hundreds or thousands of lines of Abjad's code into the global namespace for you to use. Abjad's code is organized into a collection of several dozen different score-related packages. These packages comprise hundreds of classes that model things like notes and rests and more than a thousand functions that let you do things like transpose music or change the way beams look in your score.

18.2 Inspecting output

Use `dir()` to take a look at the contents of the `iotools` package:

```
>>> dir(iotools)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', '__path__', '_documentation_section',
'clear_terminal', 'f', 'get_last_output_file_name', 'get_next_output_file_name', 'importtools', 'log', 'ly',
'p', 'pdf', 'play', 'profile_expr', 'redo', 'save_last_ly_as', 'save_last_pdf_as', 'show',
'spawn_subprocess', 'write_expr_to_ly', 'write_expr_to_pdf', 'z']
```

The `iotools` package implements I/O functions that help you work with the files you create in Abjad.

Use `iotools.ly()` to see the last LilyPond input file created in Abjad:

```
% Abjad revision 7636
% 2012-10-09 13:32

\version "2.17.3"
\language "english"
\include "/Users/trevorbaca/Documents/abjad/trunk/abjad/cfg/abjad.scm"

\header {
    tagline = \markup {  }
```

```
}

\score {
    c'4
}
```

Notice:

```
1. Abjad inserts two lines of %-prefixed comments at the top of the LilyPond files it creates.
2. Abjad includes version and language commands automatically.
3. Abjad includes a special abjad.scm file resident somewhere on your computer.
4. Abjad includes dummy LilyPond header.
5. Abjad includes a one-note score expression similar to the one you created in the last tutorial
```

When you called `show(note)` Abjad created the LilyPond input file shown above. Abjad then called LilyPond on that `.ly` file to create a PDF.

(Quit your text editor in the usual way to return to the Python interpreter.)

Now use `iotools.log()` to see the output LilyPond created as it ran:

```
GNU LilyPond 2.17.3
Processing `7721.ly'
Parsing...
Interpreting music...
Preprocessing graphical objects...
Finding the ideal number of pages...
Fitting music on 1 page...
Drawing systems...
Layout output to `7721.ps'...
Converting to `./7721.pdf'...
Success: compilation successfully completed
```

This will look familiar from the previous tutorial where we created a LilyPond file by hand.

(Quit your text editor in the usual way to return to the Python interpreter.)

CHAPTER

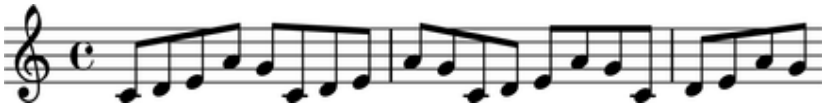
NINETEEN

CHANGING NOTES TO RESTS

19.1 A series of notes

It is easy to make a repeating pattern of notes.

```
>>> staff = Staff()
>>> staff.extend([Note(i, (1,8)) for i in [0, 2, 4, 9, 7] * 4 ])
>>> show(staff)
```



In the above example, a single list comprehension takes care of creating our notes.

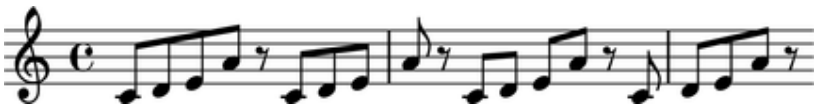
19.2 Notes belonging to a staff can be iterated

We will create our repeated pattern again. Note that you can do this in one line:

```
>>> staff = Staff([Note(i, (1,8)) for i in [0, 2, 4, 9, 7] * 4 ])
```

Now, iterate over the staff's contents, substituting an eighth rest for every fifth (count from zero!) Note element in the staff:

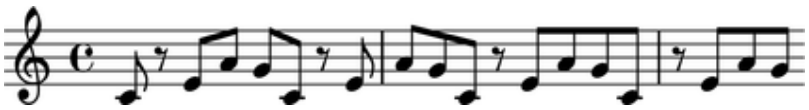
```
>>> for i, note in enumerate(staff):
...         if (i%5) == 4:
...                 staff[i] = Rest((1,8))
...
>>> show(staff)
```



19.3 Notes can be used directly

In the previous example, we used an index (i) to keep track of where we are in the list of notes, and based our decision to flip on that index. We can also decide to flip a note to a rest based on the note iteself:

```
>>> staff = Staff([Note(i, (1,8)) for i in [0, 2, 4, 9, 7] * 4 ])
>>> for i, note in enumerate(staff):
...         if note.sounding_pitch == "d'":
...                 staff[i] = Rest((1,8))
...
>>> show(staff)
```



All the d's are now rests.

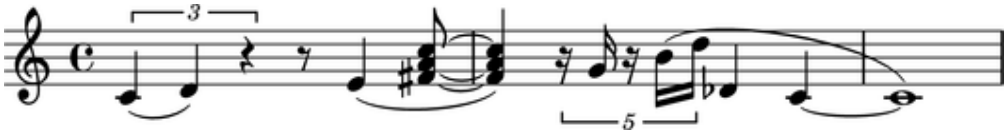
=tms

CHAPTER

TWENTY

CREATING REST-DELIMITED SLURS

Take a look at the slurs in the following example and notice that there is a pattern to how they arranged.



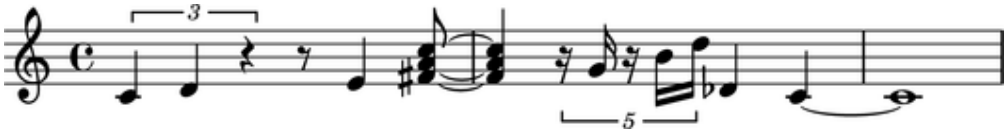
The pattern? Slurs in the example span groups of notes and chords separated by rests.

Abjad makes it easy to create rest-delimited slurs in a structured way.

20.1 Entering input

Let's start with the note input like this:

```
>>> string = r"\times 2/3 { c'4 d' r } r8 e'4 <fs' a' c''>8 ~ q4 \times 4/5 { r16 g' r b' d'' } df'4 c' ~ c'1"
>>> staff = Staff(string)
>>> show(staff)
```



20.2 Grouping notes and chords

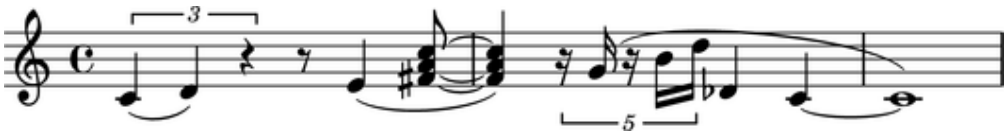
Next we'll group notes and chords together with one of the functions available in the `componenttools` package.

We add slur spanners inside our loop:

```
>>> leaves = iterationtools.iterate_leaves_in_expr(staff)
>>> for group in componenttools.yield_groups_of_mixed_klasses_in_sequence(leaves, (Note, Chord)):
...     spannertools.SlurSpanner(group)
...
SlurSpanner(c'4, d'4)
SlurSpanner(e'4, <fs' a' c''>8, <fs' a' c''>4)
SlurSpanner(g'16)
SlurSpanner(b'16, d''16, df'4, c'4, c'1)
```

Here's the result:

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



But there's a problem.

Four slur spanners were generated but only three slurs are shown.

Why? Because LilyPond ignores one-note slurs.

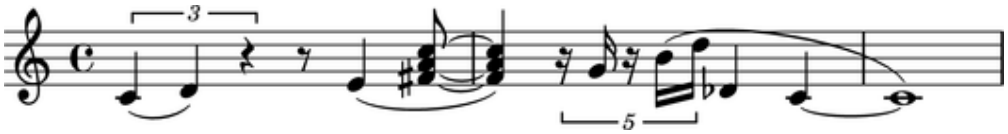
20.3 Skipping one-note slurs

Let's rewrite our example to prevent that from happening:

```
>>> staff = Staff(string)
>>> leaves = iterationtools.iterate_leaves_in_expr(staff)
>>> klasses = (Note, Chord)
>>> for group in componenttools.yield_groups_of_mixed_klasses_in_sequence(leaves, klasses):
...     if 1 < len(group):
...         spannertools.SlurSpanner(group)
...
SlurSpanner(c'4, d'4)
SlurSpanner(e'4, <fs' a' c''>8, <fs' a' c''>4)
SlurSpanner(b'16, d''16, df'4, c'4, c'1)
```

And here's the corrected result:

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



CHAPTER

TWENTYONE

MAKING GROB OVERRIDES

21.1 Grob-override component plug-ins

All Abjad containers have a grob-override plug-in:

```
>>> staff = Staff("c'4 d'4 e'4 f'4 g'4 a'4 g'2")
```

```
>>> staff.override.staff_symbol.color = 'blue'
```

```
>>> staff.override
LilyPondGrobOverrideComponentPlugIn(staff_symbol__color='blue')
```

All Abjad leaves have a grob-override plug-in, too:

```
>>> leaf = staff[-1]
```

```
>>> leaf.override.note_head.color = 'red'
>>> leaf.override.stem.color = 'red'
```

```
>>> leaf.override
LilyPondGrobOverrideComponentPlugIn(note_head__color='red', stem__color='red')
```

And so do Abjad spanners:

```
>>> slur = spannertools.SlurSpanner(staff[:])
```

```
>>> slur.override.slur.color = 'red'
```

```
>>> slur.override
LilyPondGrobOverrideComponentPlugIn(slur__color='red')
```

21.2 Grob proxies

Grob-override plug-ins contain grob proxies:

```
>>> leaf.override.note_head
LilyPondGrobProxy(color = 'red')
```

```
>>> leaf.override.stem
LilyPondGrobProxy(color = 'red')
```

21.3 Dot-chained override syntax

The's dot-chained grob override syntax shown here results from the special way that the Abjad grob-override plug-in and grob proxy set and get their attributes.

CHAPTER

TWENTYTWO

MAPPING LISTS TO RHYTHMS

Let's say you have a list of numbers that you want to convert into rhythmic notation. This is very easy to do. There are a number of related topics that are presented separately as other tutorials.

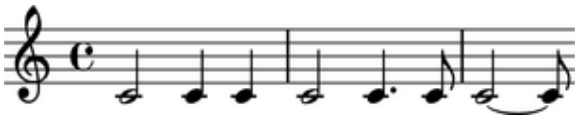
22.1 Simple example

First create a list of integer representing numerators. Then turn that list into a list of Durations instances:

```
>>> integers = [4, 2, 2, 4, 3, 1, 5]
>>> denominator = 8
>>> durations = [Duration(i, denominator) for i in integers]
```

Now we notate them using a single pitch with the function *notetools.make_notes()*:

```
>>> notes = notetools.make_notes(["c'"], durations)
>>> staff = Staff(notes)
>>> show(staff)
```



There we have it. Durations notated based on a simple list of numbers. Read the tutorials on splitting rhythms based on beats or bars in order to notate more complex duration patterns. Also, consider how changing the denominator in the Fraction above would change the series of durations.

=tms

CHAPTER

TWENTYTHREE

UNDERSTANDING LILYPOND GROBS

LilyPond models music notation as a collection of graphic objects or grobs.

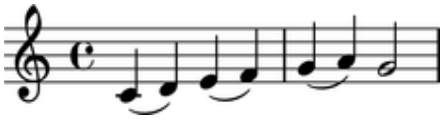
23.1 Grobs control typography

LilyPond grobs control the typographic details of the score:

```
>>> staff = Staff("c'4 ( d'4 ) e'4 ( f'4 ) g'4 ( a'4 ) g'2")
```

```
>>> f(staff)
\new Staff {
    c'4 (
    d'4 )
    e'4 (
    f'4 )
    g'4 (
    a'4 )
    g'2
}
```

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



In the example above LilyPond creates a grob for every printed glyph. This includes the clef and time signature as well as the note heads, stems and slurs. If the example included beams, articulations or an explicit key signature then LilyPond would create grobs for those as well.

23.2 Grobs can be overridden

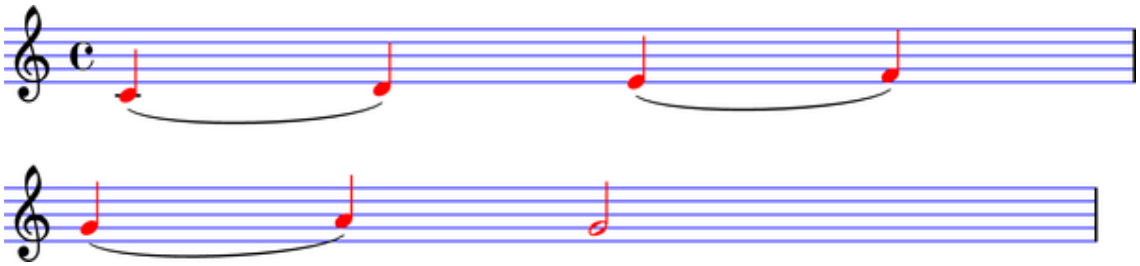
You can change the appearance of LilyPond grobs with grob overrides:

```
>>> staff.override.staff_symbol.color = 'blue'
>>> staff.override.note_head.color = 'red'
>>> staff.override.stem.color = 'red'
```

```
>>> f(staff)
\new Staff \with {
    \override NoteHead #'color = #red
    \override StaffSymbol #'color = #blue
    \override Stem #'color = #red
} {
    c'4 (
    d'4 )
    e'4 (
    f'4 )
```

```
    g'4 (
    a'4 )
    g'2
}
```

```
>>> show(staff, docs=True)
```



23.3 Nested Grob properties can be overriden

In the above example, *staff_symbol*, *note_head* and *stem* correspond to the LilyPond grobs *StaffSymbol*, *NoteHead* and *Stem*, while *color* in each case is the color properties of that graphic object.

It is not uncommon in LilyPond scores to see more complex overrides, consisting of a grob name and a list of two or more property names:

```
\override StaffGrouper #'staff-staff-spacing #'basic-distance = #7
```

To achieve the Abjad equivalent, simply concatenate the property names with double-underscores:

```
>>> staff = Staff()
>>> staff.override.staff_grouper.staff_staff_spacing__basic_distance = 7
>>> f(staff)
\new Staff \with {
    \override StaffGrouper #'staff-staff-spacing #'basic-distance = #7
} {
}
```

Abjad will explode the double-underscore delimited Python property into a LilyPond property list.

23.4 Check the LilyPond docs

New grobs are added to LilyPond from time to time.

For a complete list of LilyPond grobs see the LilyPond documentation.

CHAPTER

TWENTYFOUR

UNDERSTANDING TIME SIGNATURE MARKS

In this tutorial we take a deeper look at what happens when we attach time signature marks to staves and other score components.

At the end of the tutorial you'll understand how time signature marks are created.

You'll also understand how the states of different objects change when time signature marks are attached and detached.

24.1 Getting started

We start by creating a staff full of notes:

```
>>> staff = Staff("c'4 d'4 e'4 f'4 g'2")
```

If we ask the Abjad interpreter about our staff reference Abjad will respond with the interpreter display of the object:

```
>>> staff
Staff{5}
```

The `5` in `Staff{5}` shows that the staff contains 5 top-level components. The curly braces in `Staff{5}` show that the contents of the staff are to be read sequentially through time rather than in parallel.

Before we get to time signature marks let's take a moment and examine the state of the staff we've created. We can motivate this a bit by asking two questions:

1. what time signature is currently in effect for the staff we have just created?
2. what is the time signature currently in effect for the five notes contained within the staff we have just created?

The answer to both questions is the same: there is no time signature currently in effect for either our staff or for the five notes it contains.

We can see that this is the case with tools from the API:

```
>>> contexttools.get_effective_time_signature(staff) is None
True
```

And:

```
>>> for leaf in staff:
...     contexttools.get_effective_time_signature(leaf) is None
...
True
True
True
True
True
```

And we can iterate both the staff and its leaves at one and the same time like this:

```
>>> for component in iterationtools.iterate_components_in_expr(staff):
...     component, contexttools.get_effective_time_signature(component)
...
(Staff{5}, None)
(Note("c'4"), None)
(Note("d'4"), None)
(Note("e'4"), None)
(Note("f'4"), None)
(Note("g'2"), None)
```

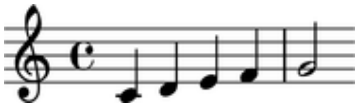
This confirms the answer to our questions that there is not yet any time signature in effect for any component in our staff because we have not yet attached a time signature mark to any component in our staff.

24.2 LilyPond's implicit `4/4`

So what happens if we format our staff and send it off to LilyPond to render as a PDF? Will LilyPond render the staff with a time signature? Without a time signature? Will LilyPond refuse to render the example at all?

We find out like this:

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



It turns out LilyPond defaults to a time signature of `4/4`.

What's important to note here is that because we have not yet attached a time signature mark any component in our staff Abjad says "no effective time signature here" while LilyPond says "OK, I'll default to `4/4` so we can get on with rendering your music."

We can further confirm that this is the case by asking Abjad for the LilyPond format of our staff:

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

The LilyPond format of our staff contains no LilyPond `\time` command. This is, again, because we have not yet attached a time signature mark to any component in our staff.

24.3 Using time signature marks

We can now practice attaching and detaching time signature marks to different components in our staff and study what happens as we do.

We'll start with `3/4`.

The easiest thing to do is to attach a time signature mark to the staff itself.

We'll do this in two separate steps and study each step to understand exactly what's going on.

First, we create a `3/4` time signature mark:

```
>>> time_signature_mark = contexttools.TimeSignatureMark((3, 4))
```

If we ask the Abjad interpreter for the interpreter dispaly of our time signature mark we get the following:

```
>>> time_signature_mark
TimeSignatureMark((3, 4))
```

All this tells us is that we have in fact created a `3/4` time signature mark. Nothing too exciting yet. At this point our `3/4` time signature is not yet attached to anything. We could say that the "state" of our time signature mark is "unattached." And we can see this like so:

```
>>> time_signature_mark.start_component is None
True
```

What does it mean for a time signature mark to have `'start_component'` equal to none? It means that the time signature isn't yet attached to any score component anywhere.

So now we attach our time signature mark to our staff:

```
>>> time_signature_mark.attach(staff)
TimeSignatureMark((3, 4))(Staff{5})
```

Abjad responds immediately by returning the time signature mark we have just attached.

Notice that our time signature mark's repr has changed. The repr of our `3/4` time signature mark now includes the repr of the staff to which we have just attached the time signature mark. That is to say that the repr of our time signature mark is statal.

Our time signature mark has transitioned from an "unattached" state to an "attached" state. We can see this like so:

```
>>> time_signature_mark.start_component
Staff{5}
```

And our staff has likewise transitioned from a state of having no effective time signature to a state of having an effective time signature:

```
>>> contexttools.get_effective_time_signature(staff)
TimeSignatureMark((3, 4))(Staff{5})
```

And what about the leaves inside our staff? Do the leaves now "know" that they are governed by a `3/4` time signature?

Indeed they do:

```
>>> for leaf in staff.leaves:
...     leaf, contexttools.get_effective_time_signature(leaf)
...
(Note("c'4"), TimeSignatureMark((3, 4))(Staff{5}))
(Note("d'4"), TimeSignatureMark((3, 4))(Staff{5}))
(Note("e'4"), TimeSignatureMark((3, 4))(Staff{5}))
(Note("f'4"), TimeSignatureMark((3, 4))(Staff{5}))
(Note("g'2"), TimeSignatureMark((3, 4))(Staff{5}))
```

So to briefly resume:

What we just did was to:

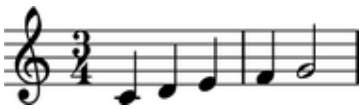
1. create a time signature mark
2. attach the time signature to a score component

This 2-step pattern is always the same when dealing with context marks: create then attach.

(We will find out later that there are short-cuts for different parts of this process. Right now we've chosen to create in a first step and attach in a second step so that we can examine the changing states of the objects involved.)

Before moving on let's look at the PDF corresponding to our staff:

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



And let's confirm what we see in the PDF in the staff's format:

```
>>> f(staff)
\new Staff {
    \time 3/4
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

The staff's format now contains a LilyPond `\time` command because we have attached an Abjad time signature mark to the staff.

What we've just been through above will cover over 80% of what you'll ever wind up doing with time signature marks: creating them and attaching them directly to staves. But what if we wanna get rid of a time signature mark? Or what if the time signature will be changing all over the place? We cover those cases next.

Detaching a time signature mark is easy:

```
>>> time_signature_mark.detach()
TimeSignatureMark((3, 4))
```

The Abjad returns the mark we have just detached. And, observing the repr of the time signature mark, we see that the time signature mark has again changed state: the time signature mark has transitioned from attached to unattached. We confirm this like so:

```
>>> time_signature_mark.start_component is None
True
```

And also like so:

```
>>> contexttools.get_effective_time_signature(staff) is None
True
```

Yup: our time signature mark knows nothing about our staff. And vice versa. This is good.

So now what if we want to set up a time signature of `2/4`? That fits our music, too.

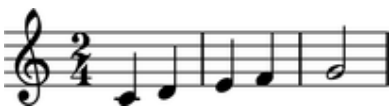
We have a couple of options.

We can simply create and attach a new time signature mark:

```
>>> duple_time_signature_mark = contexttools.TimeSignatureMark((2, 4))
>>> duple_time_signature_mark.attach(staff)
TimeSignatureMark((2, 4))(Staff{5})
```

```
>>> f(staff)
\new Staff {
    \time 2/4
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

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



Yup. That works.

On the other hand, we could simply reuse our previous `3/4` time signature mark.

To do this we'll first detach our `2/4` time signature mark ...

```
>>> duple_time_signature_mark.detach()
TimeSignatureMark((2, 4))
```

... confirm that our staff is now time signatureless ...

```
>>> contexttools.get_effective_time_signature(staff) is None
True
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

... reattach our previous `3/4` time signature ...

```
>>> time_signature_mark.attach(staff)
TimeSignatureMark((3, 4))(Staff{5})
```

... change the numerator of our time signature mark ...

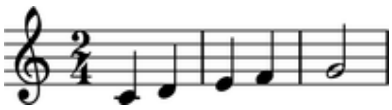
```
>>> time_signature_mark.numerator = 2
```

... and check to make sure that everything is as it should be:

```
>>> contexttools.get_effective_time_signature(staff)
TimeSignatureMark((2, 4))(Staff{5})
>>> time_signature_mark.start_component
Staff{5}
```

```
>>> f(staff)
\new Staff {
    \time 2/4
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

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



And everything works as it should.

To change to, for example, `4/4` we change just change the time signature mark's numerator again:

```
>>> time_signature_mark.numerator = 4
```

```
>>> f(staff)
\new Staff {
    \time 4/4
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

24.4 First-measure pick-ups

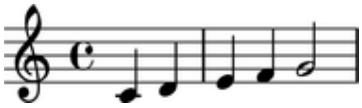
But what if our time signature has a `2/4` pick-up?

The LilyPond command for pick-ups is `\partial`. Abjad time signature marks implement this as a read / write attribute:

```
>>> time_signature_mark.partial = Duration(2, 4)
```

```
>>> f(staff)
\new Staff {
    \partial 2
    \time 4/4
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

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



And what if time signature changes all over the place?

We'll use the trivial example of a measure in `4/4` followed by a measure in `2/4`.

To do this we will need two time signature marks.

We've already got a `4/4` time signature mark attached to our staff:

```
>>> f(staff)
\new Staff {
    \partial 2
    \time 4/4
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

Let's get rid of the pick-up:

```
>>> time_signature_mark.partial = None
```

```
>>> f(staff)
\new Staff {
    \time 4/4
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

Now what about the `2/4` time signature mark?

We create it in the usual way:

```
>>> duple_time_signature_mark = contexttools.TimeSignatureMark((2, 4))
>>> duple_time_signature_mark
TimeSignatureMark((2, 4))
```

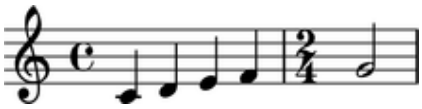
But should we attach it? We can't attach our `2/4` time signature to our staff because we've already attached our `4/4` time signature to our staff. And it only makes sense to attach one time signature to any given score component.

Observe that we've built our score in a very straightforward way: we have a single staff that contains a (flat) sequence of notes. This means that we have only one choice for where to attach the new `2/4` time signature mark. And that is one the `g'2` that comes on the downbeat of the second measure. We do that like this:

```
>>> duple_time_signature_mark.attach(staff[4])
TimeSignatureMark((2, 4))(g'2)
```

```
>>> f(staff)
\new Staff {
    \time 4/4
    c'4
    d'4
    e'4
    f'4
    \time 2/4
    g'2
}
```

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



And everything works as we would like.

Incidentally, `staff[4]` means the component sitting at index `4` inside our staff. Using the interpreter we can verify that this is `g'2`:

```
>>> staff[4]
Note("g'2")
```

Depending on how we had chosen to build our staff we would have had more options for where to attach our `2/4` time signature mark. If, for example, we had chosen to populate our staff with a series of measures then it's possible we could have attached our `2/4` time signature to a measure instead of a note.

24.5 Time signature API

That covers the vast majority of things you'll do with time signature marks.

But before we stop we should mention another useful API function and then talk about some short-cuts.

First an API function to detach ALL context marks attaching to a component:

We call the function a first time:

```
>>> contexttools.detach_context_marks_attached_to_component(staff)
(TimeSignatureMark((4, 4)),)
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
    \time 2/4
    g'2
}
```

And then a second time:

```
>>> contexttools.detach_context_marks_attached_to_component(staff[4])
(TimeSignatureMark((2, 4)),)
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
```

```
    g'2
}
```

Now there are now context marks of any sort attached to our staff or to the notes in our staff.

Be careful with this function, though: it removes *all* context marks. So even though we just used the function to remove time signature marks, it also would have removed any clef marks or tempo marks if we had had those attached to our score, too.

And now for the short-cuts:

Our staff currently has no time signature marks attached:

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

So to recreate our `3/4` time signature we can do this ...

```
>>> time_signature_mark = contexttools.TimeSignatureMark((3, 4))
```

... and then use a short-cut to avoid calling `time_signature_mark.attach()` like this:

```
>>> time_signature_mark(staff)
TimeSignatureMark((3, 4))(Staff{5})
```

```
>>> f(staff)
\new Staff {
    \time 3/4
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

What's going on here is that all context marks implement the special `__call__()` method as a short-cut for `attach()`. What is the special `__call__()` method? The `__call__()` method is what makes a function, class or any other Python object callable. The statement `time_signature_mark(staff)` has has parentheses in it because the time signature mark is callable; and the time signature mark is callable because all context marks implement the special `__call__()` method.

Note too that all context marks understand an *empty call* as a short-cut for `detach()`. Like this:

```
>>> time_signature_mark()
TimeSignatureMark((3, 4))
```

```
>>> f(staff)
\new Staff {
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

The empty call made against the time signature mark causes the time signature mark to detach from its start component.

The fact that context marks implement the special `__call__()` method as a short-cut for attach() means that context marks can be created and attached in a single line:

```
>>> contexttools.TimeSignatureMark((2, 4))(staff)
TimeSignatureMark((2, 4))(Staff{5})
```

```
>>> f(staff)
\new Staff {
    \time 2/4
    c'4
    d'4
    e'4
    f'4
    g'2
}
```

What's going on here?

What's going on is that `contexttools.TimeSignatureMark((2, 4))` creates a time signature mark in the usual way and that – immediately after this – the newly created time signature mark is available for us to call it against our staff.

This last short-cut form of ...

```
>>> contexttools.TimeSignatureMark((2, 4))(staff)
```

... is the usual way that you will see context marks of all sorts presented in the docs.

CHAPTER

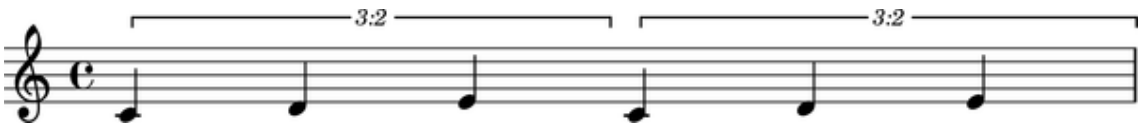
TWENTYFIVE

WORKING WITH COMPONENT PARENTAGE

Many score objects contain other score objects.

```
>>> tuplet = Tuplet(Fraction(2, 3), "c'4 d'4 e'4")
>>> staff = Staff(2 * tuplet)
>>> score = Score([staff])
```

```
>>> show(score, docs=True)
```



Abjad uses the idea of parentage to model the way objects contain each other.

25.1 Improper parentage

The improper parentage of the first note in score begins with the note itself:

```
>>> note = score.leaves[0]
```

```
>>> note.parentage
Parentage(Note("c'4"), Tuplet(2/3, [c'4, d'4, e'4]), Staff{2}, Score<<1>>)
```

```
>>> note.parentage[:]
(Note("c'4"), Tuplet(2/3, [c'4, d'4, e'4]), Staff{2}, Score<<1>>)
```

25.2 Proper parentage

The proper parentage of the note begins with only the immediate parent of the note:

```
>>> note.parentage[1:]
(Tuplet(2/3, [c'4, d'4, e'4]), Staff{2}, Score<<1>>)
```

Note: the length of the improper parentage of any component equals the length of the proper parentage of the component plus `1`.

25.3 Parentage attributes

Use Parentage to find score depth:

```
>>> note.parentage.depth
3
```

Or score root:

```
>>> note.parentage.root
Score<<1>>
```

Or to find whether a component has no (proper) parentage at all:

```
>>> note.parentage.is_orphan
False
```

CHAPTER

TWENTYSIX

WORKING WITH THREADS

26.1 What is a thread?

A thread is a structural relationship binding a set of strictly sequential voice-level components.

Threads may be explicitly defined via voice instances:

```
>>> v = Voice()
```

Or they may exist implicitly in certain score constructs in the absence of voice containers:

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

Two contiguous voices must have the same name in order to be part of the same thread.

Here a thread does **not** exist between notes in different voices:

```
>>> v_one = Voice("c'16 d'16 e'16 f'16")
>>> v_two = Voice("c'8 d'8")
>>> staff = Staff([v_one, v_two])
>>> f(staff)
\new Staff {
    \new Voice {
        c'16
        d'16
        e'16
        f'16
    }
    \new Voice {
        c'8
        d'8
    }
}
```

Here a thread does exist:

```
>>> v_one.name = 'flute'
>>> v_two.name = 'flute'
>>> f(staff)
\new Staff {
    \context Voice = "flute" {
        c'16
        d'16
        e'16
        f'16
    }
    \context Voice = "flute" {
        c'8
        d'8
    }
}
```

26.2 What are threads for?

Consider the following situation:

Are the two eighth notes in the second half of the measure the continuation of the ascending line in the first half, or is it the quarter note? Is the very last *C* the continuation of the top melodic line or is it the *A*? The stems might suggest an answer, but for Abjad, stem direction is not structural. What path should Abjad take to traverse this little score from the first note to the last *A*? This same problem appears when trying to apply spanners to parallel structures. Thus, threads are important in both score navigation and the application of spanners. In fact, threads are a requirement for spanner application.

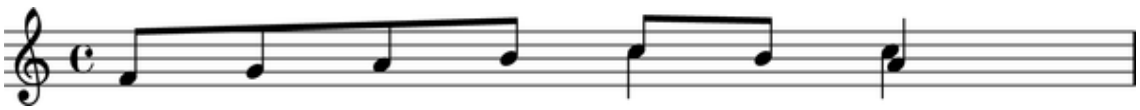
In Abjad, the ambiguity is resolved through the explicit use of named voices.

The musical fragment above is constructed with the following code:

```
>>> vA = Voice(notetools.make_notes([5, 7, 9, 11], [(1, 8)] * 4))
>>> vB = Voice(notetools.make_notes([12, 11, 9], [(1, 8), (1, 8), (1, 4)]))
>>> vC = Voice(Note(12, (1, 4)) * 2)
>>> mark = marktools.LilyPondCommandMark('voiceOne')(vA[0])
>>> mark = marktools.LilyPondCommandMark('voiceOne')(vB[0])
>>> mark = marktools.LilyPondCommandMark('voiceTwo')(vC[0])
>>> p = Container([vB, vC])
>>> p.is_parallel = True
>>> staff = Staff([vA, p])
```

```
>>> f(staff)
\new Staff {
    \new Voice {
        \voiceOne
        f'8
        g'8
        a'8
        b'8
    }
    <<
        \new Voice {
            \voiceOne
            c''8
            b'8
            a'4
        }
        \new Voice {
            \voiceTwo
            c''4
            c''4
        }
    >>
}
```

```
>>> show(staff, docs=True)
```



There's a staff that sequentially contains a voice and a parallel container. The container in turn holds two voices running simultaneously.

It is now clear from the code that the last *A* belongs with the two descending eighth notes. But there's still no indication about a relationship of continuity between the first voice in the sequence (*vA*) and any of the two following voices. Note that, while the LilyPond voice number commands setting may suggest that vA and vB belong together, this is not the case. The LilyPond voice number commands simply set the direction of stems in printed output.

To see this more clearly, suppose we want to add a slur spanner starting on the first note and ending on one of the last simultaneous notes. To attach the slur spanner to the voices we could try either:

```
>>> spannertools.SlurSpanner([vA, vB])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/media/Work/dev/scores/abjad/trunk/abjad/tools/spannertools/SlurSpanner/SlurSpanner.py", line 35, in _
    DirectedSpanner.__init__(self, components, direction)
  File "/media/Work/dev/scores/abjad/trunk/abjad/tools/spannertools/DirectedSpanner/DirectedSpanner.py", line
    Spanner.__init__(self, components)
  File "/media/Work/dev/scores/abjad/trunk/abjad/tools/spannertools/Spanner/Spanner.py", line 44, in __init__
    self._initialize_components(components)
  File "/media/Work/dev/scores/abjad/trunk/abjad/tools/spannertools/Spanner/Spanner.py", line 219, in _initial
    assert componenttools.all_are_thread_contiguous_components(leaves)
AssertionError
```

... or ...

```
>>> spannertools.SlurSpanner([vA, vC])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/media/Work/dev/scores/abjad/trunk/abjad/tools/spannertools/SlurSpanner/SlurSpanner.py", line 35, in _
    DirectedSpanner.__init__(self, components, direction)
  File "/media/Work/dev/scores/abjad/trunk/abjad/tools/spannertools/DirectedSpanner/DirectedSpanner.py", line
    Spanner.__init__(self, components)
  File "/media/Work/dev/scores/abjad/trunk/abjad/tools/spannertools/Spanner/Spanner.py", line 44, in __init__
    self._initialize_components(components)
  File "/media/Work/dev/scores/abjad/trunk/abjad/tools/spannertools/Spanner/Spanner.py", line 219, in _initial
    assert componenttools.all_are_thread_contiguous_components(leaves)
AssertionError
```

But both raise a contiguity error. Abjad needs to see an explicit connection between either *vA* and *vB* or between *vA* and *vC*.

Observe the behavior of the `iterate_thread_in_expr()` iterator on the *staff*:

```
>>> vA_thread_signature = vA.parentage.containment_signature
>>> notes = iterationtools.iterate_thread_in_expr(staff, Note, vA_thread_signature)
>>> print list(notes)
[Note("f'8"), Note("g'8"), Note("a'8"), Note("b'8")]
```

```
>>> vB_thread_signature = vB.parentage.containment_signature
>>> notes = iterationtools.iterate_thread_in_expr(staff, Note, vB_thread_signature)
>>> print list(notes)
[Note("c''8"), Note("b'8"), Note("a'4")]
```

```
>>> vC_thread_signature = vC.parentage.containment_signature
>>> notes = iterationtools.iterate_thread_in_expr(staff, Note, vC_thread_signature)
>>> print list(notes)
[Note("c''4"), Note("c''4")]
```

In each case we are passing a different **thread signature** to the `iterate_thread_in_expr()` iterator, so each case returns a different list of notes.

We can see that the thread signature of each voice is indeed different by printing it:

```
>>> vA_thread_signature = vA.parentage.containment_signature
>>> vA_thread_signature
ContainmentSignature(Voice-166691500, Voice-166691500, Staff-166692140)
```

```
>>> vB_thread_signature = vB.parentage.containment_signature
>>> vB_thread_signature
ContainmentSignature(Voice-166691628, Voice-166691628, Staff-166692140)
```

```
>>> vC_thread_signature = vC.parentage.containment_signature
>>> vC_thread_signature
ContainmentSignature(Voice-166691884, Voice-166691884, Staff-166692140)
```

And by comparing them with the binary equality operator:

```
>>> vA_thread_signature == vB_thread_signature
False
>>> vA_thread_signature == vC_thread_signature
False
>>> vB_thread_signature == vC_thread_signature
False
```

To allow Abjad to treat the content of, say, voices *vA* and *vB* as belonging together, we explicitly define a thread between them. To do this all we need to do is give both voices the same name:

```
>>> vA.name = 'piccolo'
>>> vB.name = 'piccolo'
```

Now *vA* and *vB* and all their content belong to the same thread:

```
>>> vA_thread_signature == vB_thread_signature
False
```

Note how the thread signatures have changed:

```
>>> vA_thread_signature = vA.parentage.containment_signature
>>> print vA_thread_signature
     staff: Staff-166692140
     voice: Voice-'piccolo'
      self: Voice-'piccolo'
```

```
>>> vB_thread_signature = vB.parentage.containment_signature
>>> print vB_thread_signature
     staff: Staff-166692140
     voice: Voice-'piccolo'
      self: Voice-'piccolo'
```

```
>>> vC_thread_signature = vC.parentage.containment_signature
>>> print vC_thread_signature
     staff: Staff-166692140
     voice: Voice-166691884
      self: Voice-166691884
```

And how the `iterationtools.iterate_thread_in_expr()` function returns all the notes belonging to both *vA* and *vB* when passing it the full staff and the thread signature of *vA*:

```
>>> notes = iterationtools.iterate_thread_in_expr(staff, Note, vA_thread_signature)
>>> print list(notes)
[Note("f'8"), Note("g'8"), Note("a'8"), Note("b'8"), Note("c''8"), Note("b'8"), Note("a'4")]
```

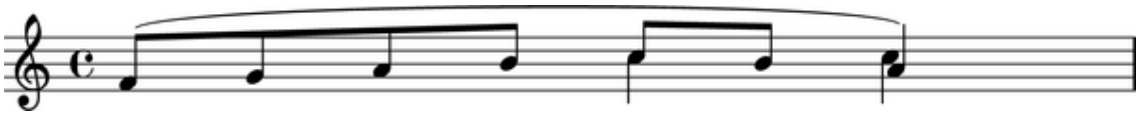
Now the slur spanner can be applied to voices *vA* and *vB*:

```
>>> spannertools.SlurSpanner([vA, vB])
SlurSpanner({f'8, g'8, a'8, b'8}, {c''8, b'8, a'4})
```

or directly to the notes returned by the `iterate_thread_in_expr()` iteration tool, which are the notes belonging to both *vA* and *vB*:

```
>>> notes = iterationtools.iterate_thread_in_expr(staff, Note, vA_thread_signature)
>>> spannertools.SlurSpanner(list(notes))
SlurSpanner(f'8, g'8, a'8, b'8, c''8, b'8, a'4)
```

```
>>> show(staff, docs=True)
```



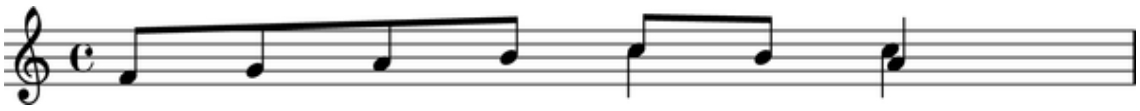
26.3 Coda

We could have constructed this score in a simpler way with only two voices, one of them starting with a LilyPond skip:

```
>>> vX = Voice(notetools.make_notes([5, 7, 9, 11, 12, 11, 9], [(1, 8)] * 6 + [(1, 4)]))
>>> vY = Voice([skiptools.Skip((2, 4))] + Note(12, (1, 4)) * 2)
>>> mark = marktools.LilyPondCommandMark('voiceOne')(vX[0])
>>> mark = marktools.LilyPondCommandMark('voiceTwo')(vY[0])
>>> staff = Staff([vX, vY])
>>> staff.is_parallel = True
```

```
>>> f(staff)
\new Staff <<
    \new Voice {
        \voiceOne
        f'8
        g'8
        a'8
        b'8
        c''8
        b'8
        a'4
    }
    \new Voice {
        \voiceTwo
        s2
        c''4
        c''4
    }
>>
```

```
>>> show(staff, docs=True)
```



Part V

Reference manual

CHAPTER

TWENTYSEVEN

ANNOTATIONS

Annotate components with user-specific information for future use.

Annotations do not impact formatting.

27.1 Creating annotations

Use mark tools to create annotations:

```
>>> annotation = marktools.Annotation('special pitch', pitchtools.NamedChromaticPitch('bs'))
```

```
>>> annotation
Annotation('special pitch', NamedChromaticPitch('bs'))
```

27.2 Attaching annotations to a component

Attach annotations to any component with `attach()`:

```
>>> note = Note("c'4")
>>> annotation.attach(note)
Annotation('special pitch', NamedChromaticPitch('bs'))(c'4)
```

```
>>> annotation
Annotation('special pitch', NamedChromaticPitch('bs'))(c'4)
```

```
>>> another_annotation = marktools.Annotation('special pitch', pitchtools.NamedChromaticPitch('bs'))
>>> another_annotation.attach(note)
Annotation('special pitch', NamedChromaticPitch('bs'))(c'4)
```

```
>>> another_annotation
Annotation('special pitch', NamedChromaticPitch('bs'))(c'4)
```

27.3 Getting the annotations attached to a component

Use mark tools to get all the annotations attached to a component:

```
>>> marktools.get_annotations_attached_to_component(note)
(Annotation('special pitch', NamedChromaticPitch('bs'))(c'4), Annotation('special pitch', NamedChromaticPitch
```

27.4 Detaching annotations from a component one at a time

Use `detach()` to detach annotations from a component one at a time:

```
>>> annotation.detach()
Annotation('special pitch', NamedChromaticPitch('bs'))
```

```
>>> annotation
Annotation('special pitch', NamedChromaticPitch('bs'))
```

27.5 Detaching all annotations attached to a component at once

Or use mark tools to detach all annotations attachd to a component at once:

```
>>> print marktools.detach_annotations_attached_to_component(note)
(Annotation('special pitch', NamedChromaticPitch('bs')),)
```

```
>>> marktools.get_annotations_attached_to_component(note)
()
```

27.6 Inspecting the component to which an annotation is attached

Use `start_component` to inspect the component to which an annotation is attached:

```
>>> annotation.attach(note)
Annotation('special pitch', NamedChromaticPitch('bs'))(c'4)
```

```
>>> annotation.start_component
Note("c'4")
```

27.7 Inspecting annotation name

Use `name` to get the name of any annotation:

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

27.8 Inspecting annotation value

And use `value` to get the value of any annotation:

```
>>> annotation.value
NamedChromaticPitch('bs')
```

CHAPTER

TWENTYEIGHT

ARTICULATIONS

Articulations model staccati, marcati, tenuti and other symbols. Articulations attach notes, rests or chords.

28.1 Creating articulations

Use `marktools` to create articulations:

```
>>> articulation = marktools.Articulation('turn')
```

```
>>> articulation
Articulation('turn')
```

28.2 Attaching articulations to a leaf

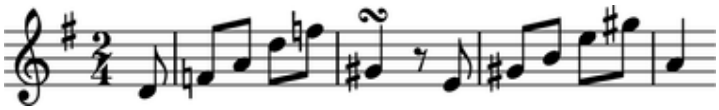
Use `attach()` to attach articulations to a leaf:

```
>>> staff = Staff([])
>>> key_signature = contexttools.KeySignatureMark('g', 'major')
>>> key_signature.attach(staff)
KeySignatureMark(NamedChromaticPitchClass('g'), Mode('major'))(Staff{})
>>> time_signature = contexttools.TimeSignatureMark((2, 4), partial = Duration(1, 8))
>>> time_signature.attach(staff)
TimeSignatureMark((2, 4), partial=Duration(1, 8))(Staff{})
```

```
>>> staff.extend("d'8 f'8 a'8 d''8 f''8 gs'4 r8 e'8 gs'8 b'8 e''8 gs''8 a'4")
```

```
>>> articulation.attach(staff[5])
Articulation('turn')(gs'4)
```

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



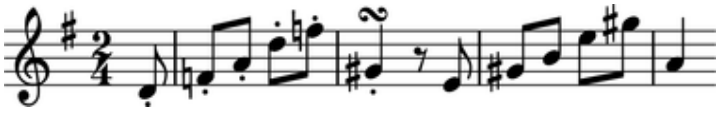
(The example is based on Haydn's piano sonata number 42, Hob. XVI/27.)

28.3 Attaching articulations to many notes and chords at once

Use `marktools` to attach articulations to many notes and chords at one time:

```
>>> marktools.attach_articulations_to_notes_and_chords_in_expr(staff[:6], ['.'])
```

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



28.4 Getting the articulations attached to a leaf

Use `marktools` to get the articulations attached to a leaf:

```
>>> marktools.get_articulations_attached_to_component(staff[5])
(Articulation('turn')(gs'4), Articulation('.')(gs'4))
```

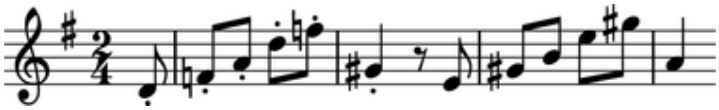
28.5 Detaching articulations from a leaf one at a time

Detach articulations by hand with `detach()`:

```
>>> articulation.detach()
Articulation('turn')
```

```
>>> articulation
Articulation('turn')
```

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



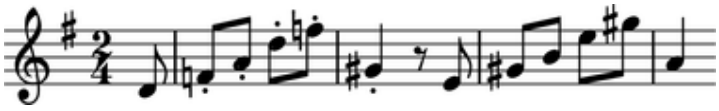
28.6 Detaching all articulations attached to a leaf at once

Use `marktools` to detach all articulations attached to a leaf at once:

```
>>> staff[0]
Note("d'8")
```

```
>>> marktools.detach_articulations_attached_to_component(staff[0])
(Articulation('.'),)
```

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

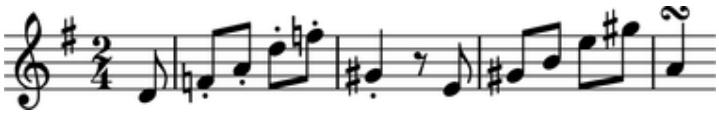


28.7 Inspecting the leaf to which an articulation is attached

Use `start_component` to inspect the component to which an articulation is attached:

```
>>> articulation = marktools.Articulation('turn')
>>> articulation.attach(staff[-1])
Articulation('turn')(a'4)
```

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



```
>>> articulation.start_component
Note("a'4")
```

28.8 Understanding the interpreter display of an articulation that is not attached to a leaf

The interpreter display of an articulation that is not attached to a leaf contains three parts:

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

```
>>> articulation
Articulation('staccato')
>>> print repr(articulation)
Articulation('staccato')
```

`Articulation` tells you the articulation's class.

`'staccato'` tells you the articulation's name.

If you set the direction string of the articulation then that will appear, too:

```
>>> articulation.direction = '^'
```

```
>>> articulation
Articulation('staccato', Up)
>>> print repr(articulation)
Articulation('staccato', Up)
```

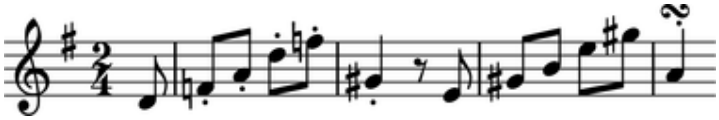
28.9 Understanding the interpreter display of an articulation that is attached to a leaf

The interpreter display of an articulation that is attached to a leaf contains four parts:

```
>>> articulation.attach(staff[-1])
Articulation('staccato', Up)(a'4)
```

```
>>> articulation
Articulation('staccato', Up)(a'4)
>>> print repr(articulation)
Articulation('staccato', Up)(a'4)
```

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



`Articulation` tells you the articulation's class.

`'staccato'` tells you the articulation's name.

`'^'` tells you the articulation's direction string.

`(a"4)` tells you the component to which the articulation is attached.

If you set the direction string of the articulation to none then the direction will no longer appear:

```
>>> articulation.direction = None
```

```
>>> articulation
Articulation('staccato')(a'4)
```

28.10 Understanding the string representation of an articulation

The string representation of an articulation comprises two parts:

```
>>> str(articulation)
'-\\staccato'
```

`-` tells you the articulation's direction string.

`staccato` tells you the articulation's name.

28.11 Inspecting the LilyPond format of an articulation

Get the LilyPond input format of an articulation with `format`:

```
>>> articulation.lilypond_format
'-\\staccato'
```

Use `f()` as a short-cut to print the LilyPond format of an articulation:

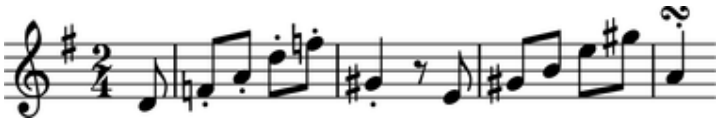
```
>>> f(articulation)
-\staccato
```

28.12 Controlling whether an articulation appears above or below the staff

Set `direction` to `'^'` to force an articulation to appear above the staff:

```
>>> articulation.direction = '^'
```

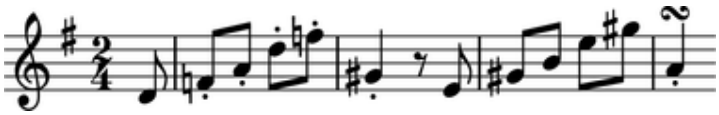
```
>>> show(staff)
```



Set `direction` to `'_'` to force an articulation to appear above the staff:

```
>>> articulation.direction = '_'
```

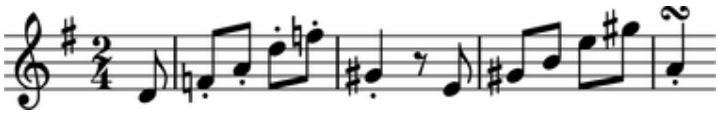
```
>>> show(staff)
```



Set `direction` to none to allow LilyPond to position an articulation automatically:

```
>>> articulation.direction = None
```

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

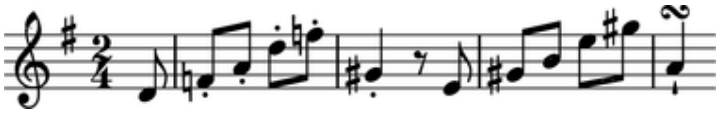


28.13 Getting and setting the name of an articulation

Set the `name` of an articulation to change the symbol an articulation prints:

```
>>> articulation.name = 'staccatissimo'
```

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



28.14 Copying articulations

Use `copy.copy()` to copy an articulation:

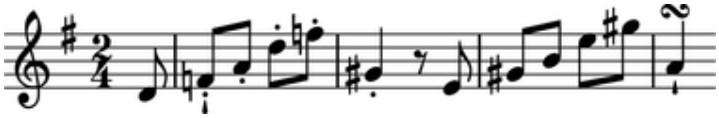
```
>>> import copy
```

```
>>> articulation_copy_1 = copy.copy(articulation)
```

```
>>> articulation_copy_1
Articulation('staccatissimo')
```

```
>>> articulation_copy_1.attach(staff[1])
Articulation('staccatissimo')(f'8)
```

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



Or use `copy.deepcopy()` to do the same thing.

28.15 Comparing articulations

Articulations compare equal with equal direction names and direction strings:

```
>>> articulation.name
'staccatissimo'
>>> articulation.direction
```

```
>>> articulation_copy_1.name
'staccatissimo'
>>> articulation_copy_1.direction
```

```
>>> articulation == articulation_copy_1
True
```

Otherwise articulations do not compare equal.

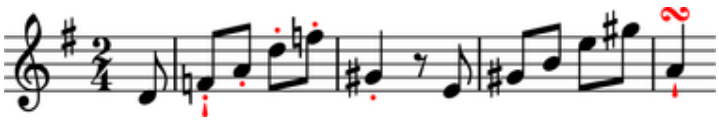
28.16 Overriding attributes of the LilyPond script grob

Override attributes of the LilyPond script grob like this:

```
>>> staff.override.script.color = 'red'
```

```
>>> f(staff)
\new Staff \with {
    \override Script #'color = #red
} {
    \key g \major
    \partial 8
    \time 2/4
    d'8
    f'8 -\staccatissimo -\staccato
    a'8 -\staccato
    d''8 -\staccato
    f''8 -\staccato
    gs'4 -\staccato
    r8
    e'8
    gs'8
    b'8
    e''8
    gs''8
    a'4 -\staccatissimo -\turn
}
```

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



See the LilyPond documentation for a list of script grob attributes available.

CHAPTER

TWENTYNINE

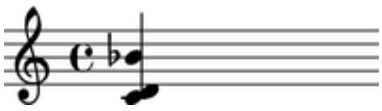
CHORDS

29.1 Making chords from a LilyPond input string

You can make chords from a LilyPond input string:

```
>>> chord = Chord("<c' d' bf'>4")
```

```
>>> show(chord, docs=True)
```

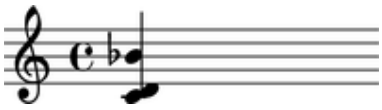


29.2 Making chords from chromatic pitch numbers and duration

You can also make chords from chromatic pitch numbers and duration:

```
>>> chord = Chord([0, 2, 10], Duration(1, 4))
```

```
>>> show(chord, docs=True)
```



29.3 Getting all the written pitches of a chord at once

You can get all the written pitches of a chord at one time:

```
>>> chord.written_pitches
(NamedChromaticPitch("c'"), NamedChromaticPitch("d'"), NamedChromaticPitch("bf'"))
```

Abjad returns a read-only tuple of named chromatic pitches.

29.4 Getting the written pitches of a chord one at a time

You can get the written pitches of a chord one at a time:

```
>>> chord.written_pitches[0]
NamedChromaticPitch("c'")
```

Chords index the pitch they contain starting from `0` (just like tuples and lists).

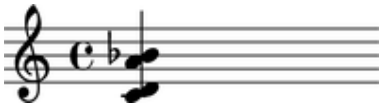
29.5 Adding one pitch to a chord at a time

Use `append()` to add one note to a chord.

You can add a pitch to a chord with a chromatic pitch number:

```
>>> chord.append(9)
```

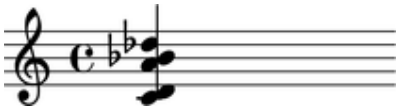
```
>>> show(chord, docs=True)
```



Or you can add a pitch to a chord with a chromatic pitch name:

```
>>> chord.append("df''")
```

```
>>> show(chord, docs=True)
```



Chords sort their pitches every time you add a new one.

This means you can add pitches to your chord in any order.

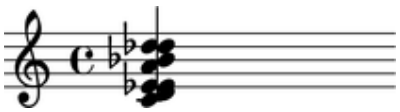
29.6 Adding many pitches to a chord at once

Use `extend()` to add many pitches to a chord.

You can use chromatic pitch numbers:

```
>>> chord.extend([3, 4, 14])
```

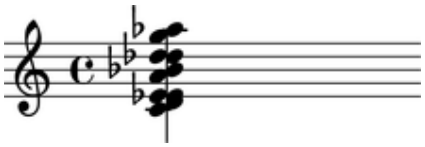
```
>>> show(chord, docs=True)
```



Or you can chromatic pitch names:

```
>>> chord.extend(["g''", "af''"])
```

```
>>> show(chord, docs=True)
```

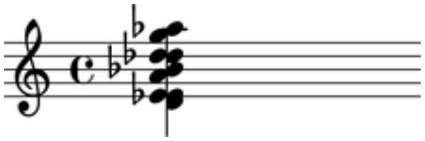


29.7 Deleting pitches from a chord

Delete pitches from a chord with `del()`:

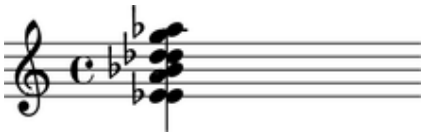
```
>>> del(chord[0])
```

```
>>> show(chord, docs=True)
```



```
>>> del(chord[0])
```

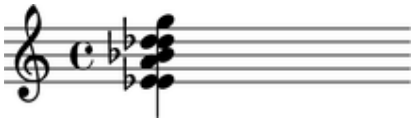
```
>>> show(chord, docs=True)
```



Negative indices work too:

```
>>> del(chord[-1])
```

```
>>> show(chord, docs=True)
```



29.8 Formatting chords

Get the LilyPond input format of any Abjad object with `format`:

```
>>> chord.lilypond_format
"<ef' e' a' bf' df'' d'' g''>4"
```

Use `f()` as a short-cut to print the LilyPond input format of any Abjad object:

```
>>> f(chord)
<ef' e' a' bf' df'' d'' g''>4
```

29.9 Working with note heads

Most of the time you will work with the pitches of a chord. But you can get the note heads of a chord, too:

```
>>> chord.note_heads
(NoteHead("ef'"), NoteHead("e'"), NoteHead("a'"), NoteHead("bf'"), NoteHead("df''"), NoteHead("d''"), NoteHead
```

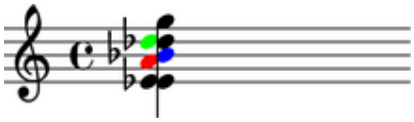
This is useful when you want to apply LilyPond overrides to note heads in a chord one at a time:

```
>>> chord[2].tweak.color = 'red'
>>> chord[3].tweak.color = 'blue'
>>> chord[4].tweak.color = 'green'
```

```
>>> f(chord)
<
    ef'
    e'
    \tweak #'color #red
    a'
    \tweak #'color #blue
    bf'
    \tweak #'color #green
    df''
```

```
    d''
    g''
>4
```

```
>>> show(chord, docs=True)
```



29.10 Working with empty chords

Abjad allows empty chords:

```
>>> chord = Chord([], Duration(1, 4))
```

Abjad formats empty chords, too:

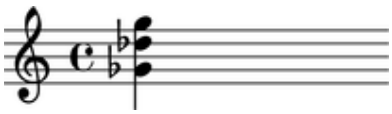
```
>>> f(chord)
<>4
```

But if you pass empty chords to `show()` LilyPond will complain because empty chords don't constitute valid LilyPond input.

When you are done working with an empty chord you can add pitches back into it chord in any of the ways described above:

```
>>> chord.extend(["gf'", "df''", "g''"])
```

```
>>> show(chord, docs=True)
```



CHAPTER

THIRTY

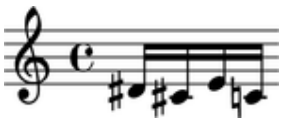
CONTAINERS

30.1 Creating containers

Create a container with components:

```
>>> container = Container([Note("ds'16"), Note("cs'16"), Note("e'16"), Note("c'16")])
```

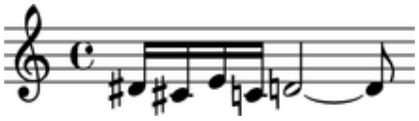
```
>>> show(container)
```



Or with a note-entry string:

```
>>> container = Container("ds'16 cs'16 e'16 c'16 d'2 ~ d'8")
```

```
>>> show(container)
```



30.2 Inspecting music

Return the components in a container with `music`:

```
>>> container.music
(Note("ds'16"), Note("cs'16"), Note("e'16"), Note("c'16"), Note("d'2"), Note("d'8"))
```

Or with a special call to `__getslice__`:

```
>>> container[:]
Selection(Note("ds'16"), Note("cs'16"), Note("e'16"), Note("c'16"), Note("d'2"), Note("d'8"))
```

30.3 Inspecting length

Get the length of a container with `len()`:

```
>>> len(container)
6
```

30.4 Inspecting duration

Contents duration equals the sum of the duration of everything inside the container:

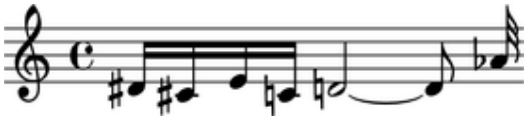
```
>>> container.contents_duration
Duration(7, 8)
```

30.5 Adding one component to the end of a container

Add one component to the end of a container with `append`:

```
>>> container.append(Note("af'32"))
```

```
>>> show(container)
```

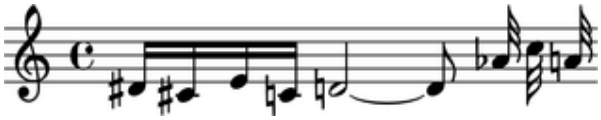


30.6 Adding many components to the end of a container

Add many components to the end of a container with `extend`:

```
>>> container.extend([Note("c''32"), Note("a'32")])
```

```
>>> show(container)
```



30.7 Finding the index of a component

Find the index of a component with `index`:

```
>>> note = container[7]
```

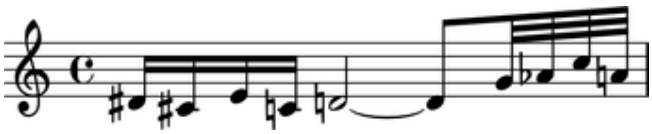
```
>>> container.index(note)
7
```

30.8 Inserting a component by index

Insert a component by index with `insert`:

```
>>> container.insert(-3, Note("g'32"))
```

```
>>> show(container)
```

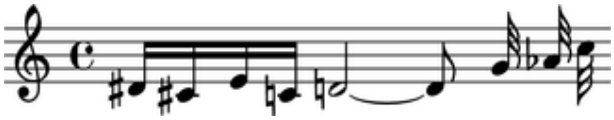


30.9 Removing a component by index

Remove a component by index with `pop`:

```
>>> container.pop(-1)
Note("a'32")
```

```
>>> show(container)
```

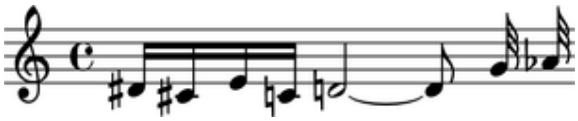


30.10 Removing a component by reference

Remove a component by reference with `remove`:

```
>>> container.remove(container[-1])
```

```
>>> show(container)
```



Note: `__getslice__`, `__setslice__` and `__delslice__` remain to be documented.

30.11 Naming containers

You can name Abjad containers:

```
>>> flute_staff = Staff("c'8 d'8 e'8 f'8")
>>> flute_staff.name = 'Flute'
>>> violin_staff = Staff("c'8 d'8 e'8 f'8")
>>> violin_staff.name = 'Violin'
>>> staff_group = scoretools.StaffGroup([flute_staff, violin_staff])
>>> score = Score([staff_group])
```

Container names appear in LilyPond input:

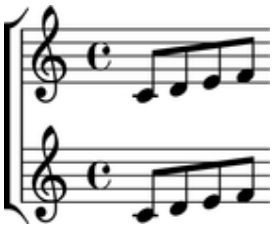
```
>>> f(score)
\new Score <<
    \new StaffGroup <<
        \context Staff = "Flute" {
            c'8
            d'8
            e'8
            f'8
        }
        \context Staff = "Violin" {
            c'8
            d'8
            e'8
            f'8
        }
    >>
>>
```

And make it easy to retrieve containers later:

```
>>> componenttools.get_first_component_in_expr_with_name(score, 'Flute')
Staff-"Flute"{4}
```

But container names do not appear in notational output:

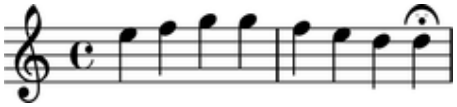
```
>>> show(score)
```



30.12 Understanding { } and << >> in LilyPond

LilyPond uses curly { } braces to wrap a stream of musical events that are to be engraved one after the other:

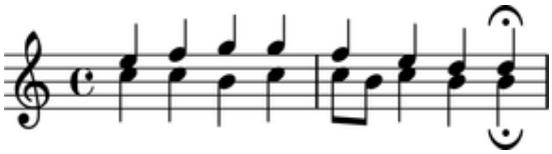
```
\new Voice {
    e''4
    f''4
    g''4
    g''4
    f''4
    e''4
    d''4
    d''4 \fermata
}
```



LilyPond uses skeleton << >> braces to wrap two or more musical expressions that are to be played at the same time:

```
\new Staff <<
    \new Voice {
        \voiceOne
        e''4
        f''4
        g''4
        g''4
        f''4
        e''4
        d''4
        d''4 \fermata
    }
    \new Voice {
        \voiceTwo
        c''4
        c''4
        b'4
        c''4
        c''8
        b'8
        c''4
        b'4
        b'4 \fermata
```

```
    }
>>
```



The examples above are both LilyPond input.

The most common use of LilyPond `{ }` is to group a potentially long stream of notes and rests into a single expression.

The most common use of LilyPond `<< >>` is to group a relatively smaller number of note lists together polyphonically.

30.13 Understanding sequential and parallel containers

Abjad implements LilyPond `{ }` and `<< >>` in the container `is_parallel` attribute.

Some containers set `is_parallel` to false at initialization:

```
staff = Staff([])
staff.is_parallel
False
```

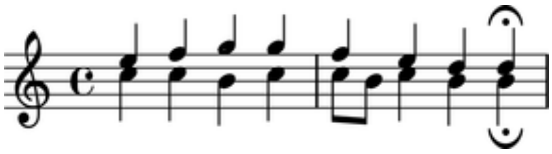
Other containers set `is_parallel` to true:

```
score = Score([])
score.is_parallel
True
```

30.14 Changing sequential and parallel containers

Set `is_parallel` by hand as necessary:

```
>>> voice_1 = Voice(r"e''4 f''4 g''4 g''4 f''4 e''4 d''4 d''4 \fermata")
>>> voice_2 = Voice(r"c''4 c''4 b'4 c''4 c''8 b'8 c''4 b'4 b'4 \fermata")
>>> staff = Staff([voice_1, voice_2])
>>> staff.is_parallel = True
>>> marktools.LilyPondCommandMark('voiceOne')(voice_1)
LilyPondCommandMark('voiceOne')(Voice{8})
>>> marktools.LilyPondCommandMark('voiceTwo')(voice_2)
LilyPondCommandMark('voiceTwo')(Voice{9})
>>> show(staff)
```



The staff in the example above is set to parallel after initialization to create a type of polyphonic staff:

```
>>> f(staff)
\new Staff <<
    \new Voice {
        \voiceOne
        e''4
        f''4
        g''4
        g''4
        f''4
```

```
        e''4
        d''4
        d''4 -\fermata
    }
    \new Voice {
        \voiceTwo
        c''4
        c''4
        b'4
        c''4
        c''8
        b'8
        c''4
        b'4
        b'4 -\fermata
    }
>>
```

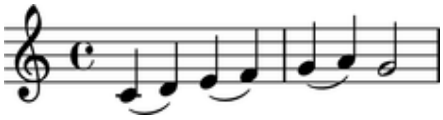
30.15 Overriding containers

The symbols below are black with fixed thickness and predetermined spacing:

```
>>> staff = Staff("c'4 d'4 e'4 f'4 g'4 a'4 g'2")
>>> slur_1 = spannertools.SlurSpanner(staff[:2])
>>> slur_2 = spannertools.SlurSpanner(staff[2:4])
>>> slur_3 = spannertools.SlurSpanner(staff[4:6])
```

```
>>> f(staff)
\new Staff {
    c'4 (
    d'4 )
    e'4 (
    f'4 )
    g'4 (
    a'4 )
    g'2
}
```

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

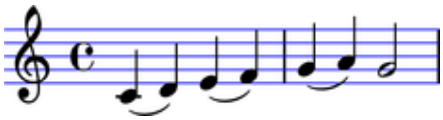


But you can override LilyPond grobs to change the look of Abjad containers:

```
>>> staff.override.staff_symbol.color = 'blue'
```

```
>>> f(staff)
\new Staff \with {
    \override StaffSymbol #'color = #blue
} {
    c'4 (
    d'4 )
    e'4 (
    f'4 )
    g'4 (
    a'4 )
    g'2
}
```

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



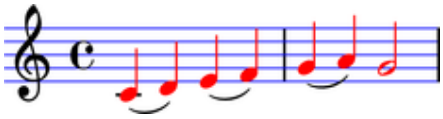
30.16 Overriding containers' contents

You can override LilyPond grobs to change the look of containers' contents, too:

```
>>> staff.override.note_head.color = 'red'
>>> staff.override.stem.color = 'red'
```

```
>>> f(staff)
\new Staff \with {
    \override NoteHead #'color = #red
    \override StaffSymbol #'color = #blue
    \override Stem #'color = #red
} {
    c'4 (
    d'4 )
    e'4 (
    f'4 )
    g'4 (
    a'4 )
    g'2
}
```

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



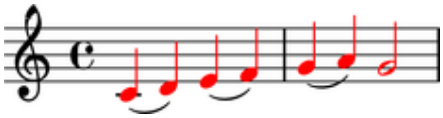
30.17 Removing container overrides

Delete grob overrides you no longer want:

```
>>> del(staff.override.staff_symbol)
```

```
>>> f(staff)
\new Staff \with {
    \override NoteHead #'color = #red
    \override Stem #'color = #red
} {
    c'4 (
    d'4 )
    e'4 (
    f'4 )
    g'4 (
    a'4 )
    g'2
}
```

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



CHAPTER

THIRTYONE

DURATIONS

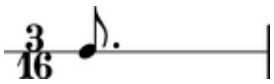
31.1 Introduction

Abjad publishes information about many durated score objects.

Notes, rests, chords and skips carry some duration attributes:

```
>>> note = Note(0, (3, 16))
>>> measure = Measure((3, 16), [note])
>>> staff = stafftools.RhythmicStaff([measure])
```

```
>>> show(staff, docs=True)
```

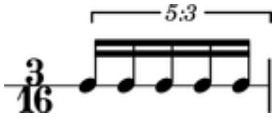


```
>>> note.written_duration
Duration(3, 16)
```

Tuplets, measures, voices, staves and the other containers carry duration attributes, too:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(3, 16), "c'16 c' c' c' c'")
>>> measure = Measure((3, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
```

```
>>> show(staff, docs=True)
```




```
>>> tuplet.multiplier
Multiplier(3, 5)
```

The next chapters document core duration concepts in Abjad.

31.2 Assignability

Western notation readily admits rational values like `1/4`. But values like `1/5` notate only with tuplet brackets or special time signatures. Abjad formalizes the difference between rationals like `1/4` and `1/5` in the definition of rational assignability.

Rational values `n/d` are assignable when and only when numerator `n` is of the form `k(2**u-j)` and denominator `d` is of the form `2**v`. In this definition `u` and `v` must be nonnegative integers, `k` must be a positive integer, and `j` must be either `0` or `1`.

Abjad initializes notes, rests and chords with assignable durations only.

31.3 Prolation

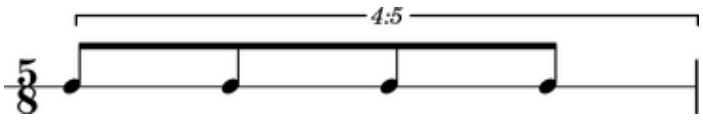
Abjad uses prolation as a cover term for rhythmic augmentation and diminution. Augmentation increases the duration of notes, rests and chords. Diminution does the opposite. Western notation employs tuplet brackets and special types of time signature to effect prolation.

31.3.1 Tuplet prolation

Tuplets prolate their contents:

```
>>> tuplet = Tuplet(Fraction(5, 4), 'c8 c c c')
>>> staff = stafftools.RhythmicStaff([Measure((5, 8), [tuplet])])
>>> beam = beamtools.BeamSpanner(tuplet)
```

```
>>> show(staff, docs=True)
```




```
>>> note = tuplet[0]
>>> note.written_duration
Duration(1, 8)
```

```
>>> note.prolation
Multiplier(5, 4)
```

```
>>> note.prolated_duration
Duration(5, 32)
```

Notes here with written duration `1/8` carry prolation factor `5/4` and prolated duration `5/32`.

Western notation does not recognize tuplet brackets carrying one-to-one ratios. Such trivial tuplets may, however, be useful during different stages of composition, and Abjad allows them for that reason. Trivial tuplets carry zero prolation. Zero-prolated tuplets neither augment nor diminish the music they contain.

31.3.2 Meter prolation

Time signatures in western notation usually carry a denominator equal to a nonnegative integer power of `2`. Abjad calls these conventional meters binary meters. Denominators equal to integers other than integer powers of `2` are also possible. Such nonbinary meters rhythmically diminish the contents of the measures they govern:

```
>>> measure = Measure((4, 10), 'c8 c c c')
>>> beam = beamtools.BeamSpanner(measure)
>>> staff = stafftools.RhythmicStaff([measure])
```

```
>>> show(staff, docs=True)
```


```
>>> note = staff.leaves[0]
>>> note.prolation
Multiplier(4, 5)
```

```
>>> note.prolated_duration
Duration(1, 10)
```

```
>>> note.prolation
Multiplier(4, 5)
```

```
>>> note.prolated_duration
Duration(1, 10)
```

Notes here with written duration `1/8` carry prolation factor `4/5` and prolated duration `1/10`.

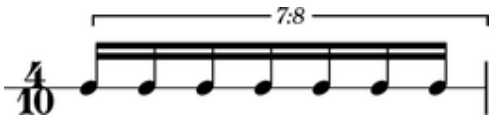
Abjad implements one of two competing nonbinary meter-interpretation schemes. The first, implicit meter-interpretation given here, follows, for example, Ferneyhough, in that nonbinary meters prolate the contents of the measures they govern implicitly, ie, without recourse to tuplet brackets. The second, explicit meter-interpretation, which we find in, for example, Sciarrino, insists instead on the presence of some tuplet bracket, usually engraved in some broken or incomplete way. The implicit meter-interpretation that Abjad implements differs from the explicit meter-interpretation native to LilyPond. Abjad will eventually implement both implicit and explicit meter-interpretation, settable on a container-by-container basis.

31.3.3 The prolation chain

Tuplets nest and combine freely with different types of meter. When two or more prolation donors conspire, the prolation factor they collectively bestow on leaf-level music equals the cumulative product of all prolation factors in the prolation chain. All durated components carry a prolation chain:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(4, 8), 'c16 c c c c c c')
>>> beamtools.BeamSpanner(tuplet)
BeamSpanner({c16, c16, c16, c16, c16, c16, c16})
>>> measure = Measure((4, 10), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
```

```
>>> show(staff, docs=True)
```




```
>>> measure.prolation
Fraction(1, 1)
```

```
>>> note = measure.leaves[0]
>>> note.prolation
Multiplier(32, 35)
```

```
>>> note.prolated_duration
Duration(2, 35)
```

Notes here with written duration `1/16` carry prolated duration `2/35`.

31.4 Duration types

Abjad publishes duration information about all score components.

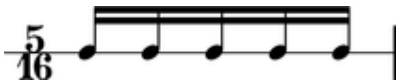
31.4.1 Written duration

Abjad uses written duration to refer to the face value of notes, rests and chords prior to prolation. Abjad written duration corresponds to the informal names most frequently used when talking about note duration.

These sixteenth notes are worth a sixteenth of a whole note:

```
>>> measure = Measure((5, 16), "c16 c c c c")
>>> beam = beamtools.BeamSpanner(measure)
>>> staff = stafftools.RhythmicStaff([measure])
```

```
>>> show(staff, docs=True)
```

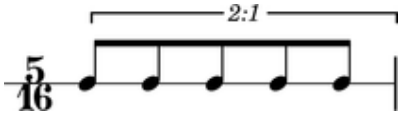


```
>>> note = measure[0]
>>> note.written_duration
Duration(1, 16)
```

These sixteenth notes are worth more than a sixteenth of a whole note:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(5, 16), "c8 c c c c")
>>> beam = beamtools.BeamSpanner(tuplet)
>>> measure = Measure((5, 16), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
```

```
>>> show(staff, docs=True)
```




```
>>> note = tuplet[0]
>>> note.written_duration
Duration(1, 8)
```

The notes in these examples are 'sixteenth notes' that carry different prolated durations. Abjad written duration captures the fact that the note heads and flag counts of the two examples match.

Written duration is a user-assignable rational number. Users can assign and reassign the written duration of notes, rests and chords at initialization and at any time during the life of the note, rest or chord. Written durations must be assignable; see the chapter on *assignability* for details. Note that Abjad containers do not carry written duration.

31.4.2 Prolated duration

Prolation refers to the duration-scaling effects of tuplets and special types of time signature. Prolation is a way of thinking about the contribution that musical structure makes to the duration of score objects. All durated Abjad objects carry a prolated duration. Prolated duration is an emergent property of notes, tuplets and other durated objects. The prolated duration of notes, rests and chords equals the product of the written duration and prolation of those objects. The prolated duration of tuplets, measures and other containers equals the the container's duration interface multiplied by the container's prolation.

31.4.3 Contents duration

Abjad defines the contents duration of tuplets, measures, voices, staves and other containers equal to the sum of the preprolated duration of each of the elements in the container.

The measure here contains two eighth notes and tuplet. These elements carry preprolated durations equal to `1/8`, `1/8` and `2/8`, respectively:

```
>>> notes = 2 * Note("c'8")
>>> beam = beamtools.BeamSpanner(notes)
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(2, 8), "c'8 c c")
>>> beam = beamtools.BeamSpanner(tuplet)
>>> measure = Measure((4, 8), notes + [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
```

```
>>> show(staff, docs=True)
```

```
>>> measure.contents_duration
Duration(1, 2)
```

The contents duration of the measure here equals `1/8 + 1/8 + 2/8 = 4/8`.

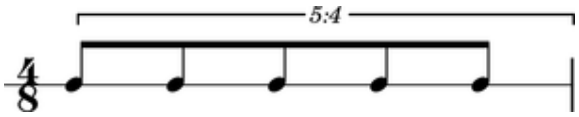
31.4.4 Target duration

Abjad defines the target duration of fixed-duration tuplets equal to composer-settable duration to which the tuplet prolates its contents.

This fixed-duration tuplet carries a target duration equal to `4/8`:

```
>>> tuplet = tuplettools.FixedDurationTuplet(Duration(4, 8), "c'8 c c c c")
>>> beam = beamtools.BeamSpanner(tuplet)
>>> measure = Measure((4, 8), [tuplet])
>>> staff = stafftools.RhythmicStaff([measure])
```

```
>>> show(staff, docs=True)
```




```
>>> tuplet.target_duration
Duration(1, 2)
```

The tuplet contents sum to `5/8`. But tuplet target duration always equals `4/8`.

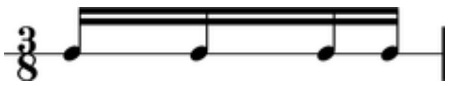
31.4.5 Multiplied duration

Abjad defines the multiplied duration of notes, rests and chords equal to the product of written duration and leaf multiplier.

The first two notes below carry leaf mulitipliers equal to `2/1`:

```
>>> notes = 4 * Note("c'16")
>>> notes[0].duration_multiplier = Fraction(2, 1)
>>> notes[1].duration_multiplier = Fraction(2, 1)
>>> measure = Measure((3, 8), notes)
>>> beam = beamtools.BeamSpanner(measure)
>>> staff = stafftools.RhythmicStaff([measure])
```

```
>>> show(staff, docs=True)
```



```
>>> note = measure[0]
>>> note.written_duration
Duration(1, 16)
```

```
>>> note.duration_multiplier
Multiplier(2, 1)
```

```
>>> note.written_duration * note.duration_multiplier
Duration(1, 8)
>>> note.multiplied_duration
Duration(1, 8)
```

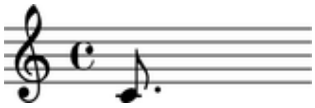
The written duration of these first two notes equals `1/16` and so the multiplied duration of these first two notes equals `1/16 * 2/1 = 1/8`.

31.5 Duration initialization

Durated Abjad classes initialize duration from arguments in the form *(n, d)* with numerator *n* and denominator *d*.

```
>>> note = Note("c'8.")
```

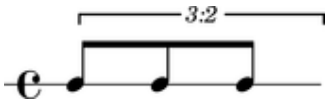
```
>>> show(note, docs=True)
```



Durated classes include notes, rests, chords, skips, tuplets and measures.

```
>>> tuplet = tuplettools.Tuplet((2, 3), "c'8 c'8 c'8")
>>> beamtools.BeamSpanner(tuplet)
BeamSpanner({c'8, c'8, c'8})
>>> staff = stafftools.RhythmicStaff([tuplet])
```

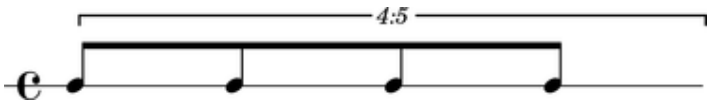
```
>>> show(staff, docs=True)
```



Abjad restricts notes, rests, chords and skips to durations like `3/16` that can be written with dots, beams and flags without ties or brackets. Abjad allows arbitrary positive durations like `5/8` for tuplets and measures.

```
>>> tuplet = tuplettools.Tuplet((5, 4), "c'8 c'8 c'8 c'8")
>>> beamtools.BeamSpanner(tuplet)
BeamSpanner({c'8, c'8, c'8, c'8})
>>> staff = stafftools.RhythmicStaff([tuplet])
```

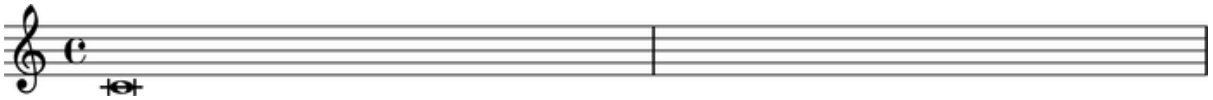
```
>>> show(staff, docs=True)
```



Abjad supports breves.

```
>>> note = Note(0, (2, 1))
```

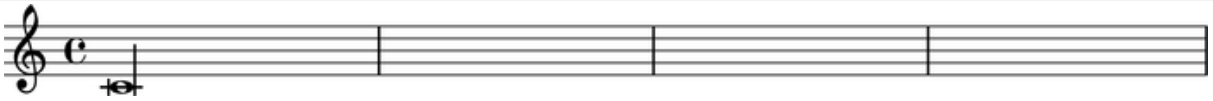
```
>>> show(note, docs=True)
```



And longas.

```
>>> note = Note(0, (4, 1))
```

```
>>> show(note, docs=True)
```



Note: The restriction that the written durations of notes, rests, chords and skips be expressible with some combination of dots, flags and beams without recourse to ties and brackets generalizes to the condition of note_head assignability. Values *(n, d)* are note_head-assignable when and only when (1) d is a nonnegative integer power of 2; (2) n is either a nonnegative integer power of 2 or is a nonnegative integer power of 2, minus 1; and (3) n/d is less than or equal to 8. Condition (3) captures the fact that LilyPond provides no glyph with greater duration than the maxima (equal to eight whole notes).

Note: Integer forms like `4` as a substitute for `(4, 1)` in `Note(0, (4, 1))` are undocumented but allowed.

Note: Abjad allows maxima note heads as in `Note(0, (8, 1))`. LilyPond implements a `\maxima` command but does not supply a corresponding glyph for the note head.

31.6 LilyPond multipliers

LilyPond provides an asterisk * operator to scale the durations of notes, rests and chords by arbitrarily positive rational values. LilyPond multipliers are inivisible and generate no typographic output of their own. However, while independent from the typographic output, LilyPond multipliers do factor in in calculations of duration and time.

Abjad implements LilyPond multpliers as the settable *duration.multiplier* attribute of notes, rests and chords.

```
>>> note = Note("c'4")
>>> note.duration_multiplier = Fraction(1, 2)
>>> note.duration_multiplier
Multiplier(1, 2)
```

```
>>> f(note)
c'4 * 1/2
```

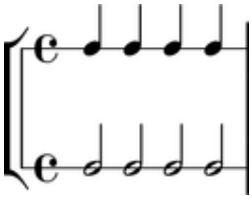
Abjad also implements a *duration.multiplied* attribute to examine the duration of a note, rest or chord as affected by the multiplier.

```
>>> note.multiplied_duration
Duration(1, 8)
```

LilyPond multipliers give the half notes here multiplied durations equal to a quarter note.

```
>>> notes = Note("c'4") * 4
>>> multiplied_note = Note(0, (1, 2))
>>> multiplied_note.duration_multiplier = Fraction(1, 2)
>>> multiplied_notes = multiplied_note * 4
>>> top = stafftools.RhythmicStaff(notes)
>>> bottom = stafftools.RhythmicStaff(multiplied_notes)
>>> staves = scoretools.StaffGroup([top, bottom])
```

```
>>> show(staves)
```



Note: Abjad models multiplication fundamentally differently than prolation . See the chapter on *Prolation* for more information.

Note: The LilyPond multiplication * operator differs from the Abjad multiplication * operator. LilyPond multiplication scales duration of LilyPond notes, rests and chords. Abjad multiplication copies Abjad containers and leaves.

31.7 Duration interfaces compared

type	core	leaf	container	measure	tuplet	fd tuplet	fm tuplet
contents	–	–	R	R	R	R	R
multiplied	–	R	–	–	–	R	R
multiplier	–	RW	–	R	R	R	RW
preprolated	R	R	R	R	R	R	R
prolated	R	R	R	R	R	R	R
prolation	R	R	R	R	R	R	R
target	–	–	–	–	–	RW	–
written	–	RW	–	–	–	–	–

The table contains a total of only four settable duration attributes, divided among only three classes. Durated Abjad classes offer up many read-only duration attributes but very few read-write duration attributes.

All classes carry all three prolation-related attributes because all classes can nest inside containers. It is possible, for example, to nest an entire voice within a fixed-duration tuplet.

Note: Leaf multipliers and tuplet multipliers differ.

CHAPTER

THIRTYTWO

INSTRUMENT MARKS

Instrument marks appear as markup in the left margin of your score.

32.1 Creating instrument marks

Use `contexttools` to create instrument marks:

```
>>> instrument_mark = contexttools.InstrumentMark('Violin ', 'Vn. ')
```

```
>>> instrument_mark
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')
```

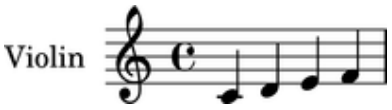
32.2 Attaching instrument marks to a component

Use `attach()` to attach any mark to a component:

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

```
>>> instrument_mark.attach(staff)
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')(Staff{4})
```

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



32.3 Getting the instrument mark attached to a component

Use `contexttools` to get the instrument mark attached to a component:

```
>>> contexttools.get_instrument_mark_attached_to_component(staff)
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')(Staff{4})
```

32.4 Getting the instrument in effect for a component

Or to get the instrument currently in effect for a component:

```
>>> contexttools.get_effective_instrument(staff[1])
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')(Staff{4})
```

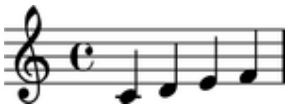
32.5 Detaching instrument marks from a component one at a time

Use `detach()` to detach instrument marks from a component one at a time:

```
>>> instrument_mark.detach()
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')
```

```
>>> instrument_mark
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')
```

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



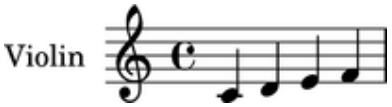
32.6 Detaching all instrument marks attached to a component at once

Or use `contexttools` to detach instrument marks all at once:

```
>>> instrument_mark = contexttools.InstrumentMark('Violin ', 'Vn. ')
>>> instrument_mark.attach(staff)
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')(Staff{4})
```

```
>>> instrument_mark
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')(Staff{4})
```

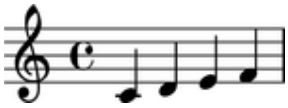
```
>>> show(staff)
```




```
>>> contexttools.detach_instrument_marks_attached_to_component(staff)
(InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. '),)
```

```
>>> instrument_mark
InstrumentMark(instrument_name='Violin ', short_instrument_name='Vn. ')
```

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

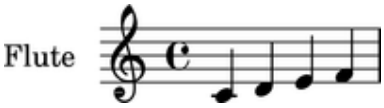


32.7 Inspecting the component to which an instrument mark is attached

Use `start_component` to inspect the component to which an instrument mark is attached:

```
>>> instrument_mark = contexttools.InstrumentMark('Flute ', 'Fl. ')
>>> instrument_mark.attach(staff)
InstrumentMark(instrument_name='Flute ', short_instrument_name='Fl. ')(Staff{4})
```

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




```
>>> instrument_mark.start_component
Staff{4}
```

32.8 Inspecting the instrument name of an instrument mark

Use `instrument_name_markup` to get the instrument name of any instrument mark:

```
>>> instrument_mark.instrument_name_markup
Markup(('Flute',))
```

32.9 Inspecting the short instrument name of an instrument mark

And use `short_instrument_name_markup` to get the short instrument name of any instrument mark:

```
>>> instrument_mark.short_instrument_name_markup
Markup(('Fl.',))
```

CHAPTER

THIRTYTHREE

I/O

33.1 Reopening Abjad PDFs

After you build a piece of notation and open with `show()` you will usually close the resulting PDF and continue working, changing your output notation in an iterative and incremental way.

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

But what if you need to go back and open the resulting PDF again? Abjad provides `pdf()` for precisely this purpose. Type the following at the Abjad prompt to open the most recent PDF written by Abjad.

```
>>> pdf()
```

If you want to open not the next-to-most recent PDF generated by Abjad, pass in a `-1`. And for the next-to-next-to-most recent, pass in a `-2`, and so on.

33.2 Looking at LilyPond output

Abjad generates a LilyPond `.ly` file for every Abjad expression that you build and `show()`. To look at these LilyPond `.ly` files that Abjad builds behind the scenes, use `ly()`.

```
>>> ly()
```

```
% Abjad revision 2362
% 2009-06-25 10:30

\version "2.12.2"
\include "english.ly"
\include "/Users/trevorbaca/Documents/abjad/trunk/abjad/scm/abjad.scm"

\new Staff {
   c'8
   d'8
   e'8
   f'8
   g'8
   a'8
   b'8
   c''8
}
```

Abjad opens the LilyPond `.ly` file in your favorite text editor.

These LilyPond `.ly` files that Abjad generates all have the same basic structure. The current version of Abjad and the date appear first, followed by the mandatory LilyPond version string and LilyPond directives for English

note names and the default Abjad `.scm` file. The remainder of the file is reserved for the LilyPond input code corresponding to the expression you just built in Abjad.

When you are done looking at the LilyPond `.ly` file quit your text editor to return to the Abjad interpreter.

33.3 Looking at the LilyPond log

If things go wrong when you call `show()` or one of the other Abjad functions that call LilyPond behind the scenes, if may be helpful to examine the output that LilyPond writes to the LilyPond log.

```
>>> log()
```

```
GNU LilyPond 2.12.2
Processing `1420.ly'
Parsing...
Interpreting music...
Preprocessing graphical objects...
Finding the ideal number of pages...
Fitting music on 1 page...
Drawing systems...
Layout output to `1420.ps'...
Converting to `./1420.pdf'...
```

This is the normal output that LilyPond generates every time you call the program behind. When you are done looking at the LilyPond log, quit your text editor to return to the Abjad interpreter.

CHAPTER

THIRTYFOUR

LILYPOND COMMAND MARKS

LilyPond command marks allow you to attach arbitrary LilyPond commands to Abjad score components.

34.1 Creating LilyPond command marks

Use `marktools` to create LilyPond command marks:

```
>>> lilypond_command_mark = marktools.LilyPondCommandMark('bar "||"', 'after')
```

```
>>> lilypond_command_mark
LilyPondCommandMark('bar "||"')
```

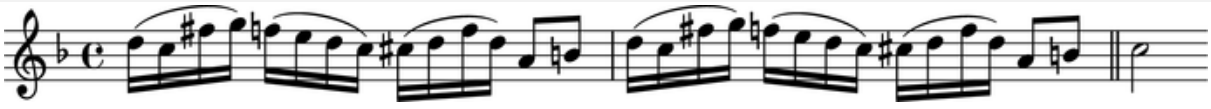
34.2 Attaching LilyPond command marks to Abjad components

Use `attach()` to attach a LilyPond command mark to any Abjad component:

```
>>> import copy
>>> staff = Staff([])
>>> key_signature = contexttools.KeySignatureMark('f', 'major')
>>> key_signature.attach(staff)
KeySignatureMark(NamedChromaticPitchClass('f'), Mode('major'))(Staff{})
>>> staff.extend(p("{ d''16 ( c''16 fs''16 g''16 ) }"))
>>> staff.extend(p("{ f''16 ( e''16 d''16 c''16 ) }"))
>>> staff.extend(p("{ cs''16 ( d''16 f''16 d''16 ) }"))
>>> staff.extend(p("{ a'8 b'8 }"))
>>> staff.extend(p("{ d''16 ( c''16 fs''16 g''16 )} "))
>>> staff.extend(p("{ f''16 ( e''16 d''16 c''16 ) }"))
>>> staff.extend(p("{ cs''16 ( d''16 f''16 d''16 ) }"))
>>> staff.extend(p("{ a'8 b'8 c''2 }"))
```

```
>>> lilypond_command_mark.attach(staff[-2])
LilyPondCommandMark('bar "||"')(b'8)
```

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



34.3 Getting the LilyPond command marks attached to an Abjad component

Use `marktools` to get the lilypond_command_marks attached to a leaf:

```
>>> marktools.get_lilypond_command_marks_attached_to_component(staff[-2])
(LilyPondCommandMark('bar "||"')(b'8),)
```

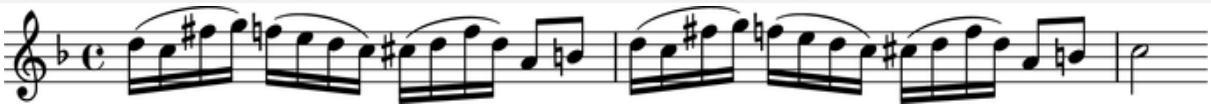
34.4 Detaching LilyPond command marks from components one at a time

Use `detach()` to detach LilyPond command marks one at a time:

```
>>> lilypond_command_mark.detach()
LilyPondCommandMark('bar "||"')
```

```
>>> lilypond_command_mark
LilyPondCommandMark('bar "||"')
```

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



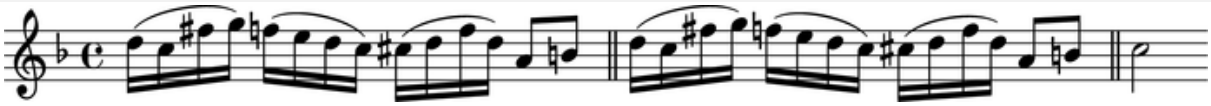
34.5 Detaching all LilyPond command marks attached to a component at once

Use `marktools` to detach all LilyPond command marks attached to a component at once:

```
>>> lilypond_command_mark_1 = marktools.LilyPondCommandMark('bar "||"', 'closing')
>>> lilypond_command_mark_1.attach(staff[-2])
LilyPondCommandMark('bar "||"')(b'8)
```

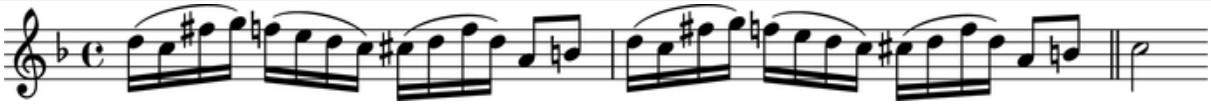
```
>>> lilypond_command_mark_2 = marktools.LilyPondCommandMark('bar "||"', 'closing')
>>> lilypond_command_mark_2.attach(staff[-16])
LilyPondCommandMark('bar "||"')(b'8)
```

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



```
>>> marktools.detach_lilypond_command_marks_attached_to_component(staff[-16])
(LilyPondCommandMark('bar "||"'),)
```

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

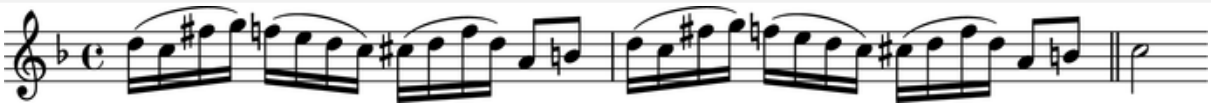


34.6 Inspecting the component to which a LilyPond command mark is attached

Use `start_component` to inspect the component to which a LilyPond command mark is attached:

```
>>> lilypond_command_mark = marktools.LilyPondCommandMark('bar "||"', 'closing')
>>> lilypond_command_mark.attach(staff[-2])
LilyPondCommandMark('bar "||"')(b'8)
```

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



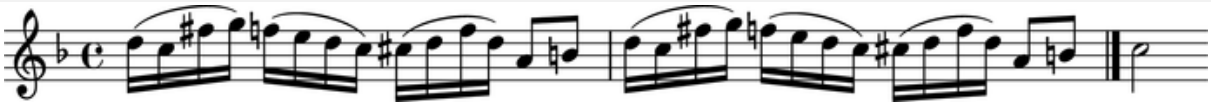
```
>>> lilypond_command_mark.start_component
Note("b'8")
```

34.7 Getting and setting the command name of a LilyPond command mark

Set the `command_name` of a LilyPond command mark to change the LilyPond command a LilyPond command mark prints:

```
>>> lilypond_command_mark.command_name = 'bar "|."'
```

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



34.8 Copying LilyPond commands

Use `copy.copy()` to copy a LilyPond command mark:

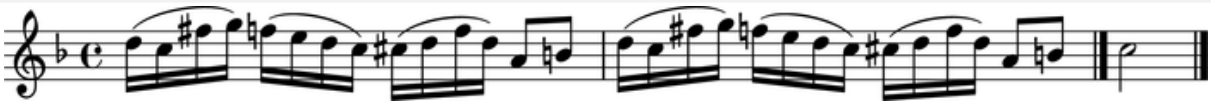
```
>>> import copy
```

```
>>> lilypond_command_mark_copy_1 = copy.copy(lilypond_command_mark)
```

```
>>> lilypond_command_mark_copy_1
LilyPondCommandMark('bar "|."')
```

```
>>> lilypond_command_mark_copy_1.attach(staff[-1])
LilyPondCommandMark('bar "|."')(c''2)
```

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



Or use `copy.deepcopy()` to do the same thing.

34.9 Comparing LilyPond command marks

LilyPond command marks compare equal with equal command names:

```
>>> lilypond_command_mark.command_name
'bar "|."'
```

```
>>> lilypond_command_mark_copy_1.command_name
'bar "|."'
```

```
>>> lilypond_command_mark == lilypond_command_mark_copy_1
True
```

Otherwise LilyPond command marks do not compare equal.

CHAPTER

THIRTYFIVE

LILYPOND COMMENTS

LilyPond comments begin with the `%` sign. Abjad models LilyPond comments as marks.

35.1 Creating LilyPond comments

Use `marktools` to create LilyPond comments:

```
>>> comment_1 = marktools.LilyPondComment('This is a LilyPond comment before a note.', 'before')
```

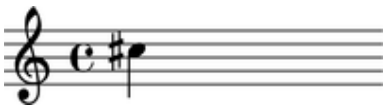
```
>>> comment_1
LilyPondComment('This is a LilyPond comment before a note.')
```

35.2 Attaching LilyPond comments to leaves

Attach LilyPond comments to a note, rest or chord with `attach()`:

```
>>> note = Note("cs''4")
```

```
>>> show(note, docs=True)
```



```
>>> comment_1.attach(note)
LilyPondComment('This is a LilyPond comment before a note.')(cs''4)
```

```
>>> f(note)
% This is a LilyPond comment before a note.
cs''4
```

You can add LilyPond comments before, after or to the right of any leaf.

35.3 Attaching LilyPond comments to containers

Use `attach()` to attach LilyPond comments to a container:

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

```
>>> show(staff, docs=True)
```

```
>>> staff_comment_1 = marktools.LilyPondComment('Here is a LilyPond comment before the staff.', 'before')
>>> staff_comment_2 = marktools.LilyPondComment('Here is a LilyPond comment in the staff opening.', 'opening')
>>> staff_comment_3 = marktools.LilyPondComment('Here is another LilyPond comment in the staff opening.', 'ope
>>> staff_comment_4 = marktools.LilyPondComment('LilyPond comment in the staff closing.', 'closing')
>>> staff_comment_5 = marktools.LilyPondComment('LilyPond comment after the staff.', 'after')
```

```
>>> staff_comment_1.attach(staff)
LilyPondComment('Here is a LilyPond comment before the staff.')(Staff{4})
>>> staff_comment_2.attach(staff)
LilyPondComment('Here is a LilyPond comment in the staff opening.')(Staff{4})
>>> staff_comment_3.attach(staff)
LilyPondComment('Here is another LilyPond comment in the staff opening.')(Staff{4})
>>> staff_comment_4.attach(staff)
LilyPondComment('LilyPond comment in the staff closing.')(Staff{4})
>>> staff_comment_5.attach(staff)
LilyPondComment('LilyPond comment after the staff.')(Staff{4})
```

```
>>> f(staff)
% Here is a LilyPond comment before the staff.
\new Staff {
    % Here is a LilyPond comment in the staff opening.
    % Here is another LilyPond comment in the staff opening.
    c'8
    d'8
    e'8
    f'8
    % LilyPond comment in the staff closing.
}
% LilyPond comment after the staff.
```

You can add LilyPond comments before, after, in the opening or in the closing of any container.

35.4 Getting the LilyPond comments attached to a component

Use `marktools` to get all the LilyPond comments attached to a component:

```
>>> marktools.get_lilypond_comments_attached_to_component(note)
(LilyPondComment('This is a LilyPond comment before a note.')(cs''4),)
```

Abjad returns a tuple of zero or more LilyPond comments.

35.5 Detaching LilyPond comments from a component one at a time

Use `detach()` to detach LilyPond comments from a component one at a time:

```
>>> comment_1 = marktools.get_lilypond_comments_attached_to_component(note)[0]
```

```
>>> comment_1.detach()
LilyPondComment('This is a LilyPond comment before a note.')
```

```
>>> f(note)
cs''4
```

35.6 Detaching all LilyPond comments attached to a component at once

Or use `marktools` to detach all LilyPond comments attached to a component at once:

```
>>> for comment in marktools.get_lilypond_comments_attached_to_component(staff): print comment
...
LilyPondComment('Here is a LilyPond comment before the staff.')(Staff{4})
LilyPondComment('Here is a LilyPond comment in the staff opening.')(Staff{4})
LilyPondComment('Here is another LilyPond comment in the staff opening.')(Staff{4})
LilyPondComment('LilyPond comment in the staff closing.')(Staff{4})
LilyPondComment('LilyPond comment after the staff.')(Staff{4})
```

```
>>> marktools.detach_lilypond_comments_attached_to_component(staff)
(LilyPondComment('Here is a LilyPond comment before the staff.'), LilyPondComment('Here is a LilyPond comment
```

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

35.7 Inspecting the component to which a LilyPond comment is attached

Use `start_component` to inspect the component to which a LilyPond comment is attached:

```
>>> comment_1.attach(note)
LilyPondComment('This is a LilyPond comment before a note.')(cs''4)
```

```
>>> comment_1.start_component
Note("cs''4")
```

35.8 Inspecting contents string of a LilyPond comment

Use `contents_string` to inspect the written contents of a LiliyPond comment:

```
>>> comment_1.contents_string
'This is a LilyPond comment before a note.'
```

CHAPTER

THIRTYSIX

LILYPOND FILES

36.1 Making LilyPond files

Make a basic LilyPond input file with the `lilypondfiletools` package:

```
>>> staff = Staff("c'8 d'8 e'8 f'8")
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(staff)
```

```
>>> lilypond_file
LilyPondFile(Staff{4})
```

36.2 Inspecting file output

LilyPond input files that you create this way come equipped with many attributes that appear in file output:

```
>>> f(lilypond_file)
% Abjad revision 8302:8306
% 2012-12-15 16:27

\version "2.16.1"
\language "english"
\include "/media/Work/dev/scores/abjad/trunk/abjad/cfg/abjad.scm"

\score {
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
}
```

36.3 Setting default paper size

Set default LilyPond paper size like this:

```
>>> lilypond_file.default_paper_size = '11x17', 'landscape'
```

```
>>> f(lilypond_file)
% Abjad revision 8302:8306
% 2012-12-15 16:27

\version "2.16.1"
\language "english"
\include "/media/Work/dev/scores/abjad/trunk/abjad/cfg/abjad.scm"

#(set-default-paper-size "11x17" 'landscape)
```

```
\score {
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
}
```

36.4 Setting global staff size

Set global staff size like this:

```
>>> lilypond_file.global_staff_size = 16
```

```
>>> f(lilypond_file)
% Abjad revision 8302:8306
% 2012-12-15 16:27

\version "2.16.1"
\language "english"
\include "/media/Work/dev/scores/abjad/trunk/abjad/cfg/abjad.scm"

#(set-default-paper-size "11x17" 'landscape)
#(set-global-staff-size 16)

\score {
    \new Staff {
        c'8
        d'8
        e'8
        f'8
    }
}
```

CHAPTER

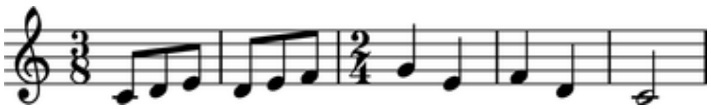
THIRTYSEVEN

MEASURES

37.1 Understanding measures in LilyPond

In LilyPond you specify time signatures by hand and LilyPond creates measures automatically:

```
\new Staff {
    \time 3/8
    c'8
    d'8
    e'8
    d'8
    e'8
    f'8
    \time 2/4
    g'4
    e'4
    f'4
    d'4
    c'2
}
```



Here LilyPond creates five measures from two time signatures. This happens because behind-the-scenes LilyPond time-keeping tells the program when measures start and stop and how to draw the barlines that come between them.

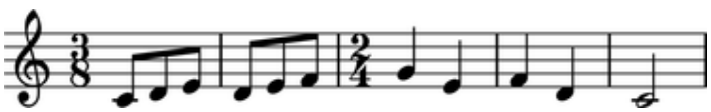
37.2 Understanding measures in Abjad

Measures are optional in Abjad, too, and you may omit them in favor of time signatures:

```
>>> staff = Staff("c'8 d'8 e'8 d'8 e'8 f'8 g'4 e'4 f'4 d'4 c'2")
```

```
>>> contexttools.TimeSignatureMark((3, 8))(staff)
TimeSignatureMark((3, 8))(Staff{11})
>>> contexttools.TimeSignatureMark((2, 4))(staff[6])
TimeSignatureMark((2, 4))(g'4)
```

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



But you may also include explicit measures in the Abjad scores you build. The following sections explain how.

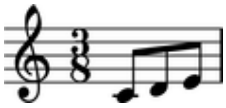
37.3 Creating measures

Create a measure with a time signature and music:

```
>>> measure = Measure((3, 8), "c'8 d'8 e'8")
```

```
>>> f(measure)
{
    \time 3/8
    c'8
    d'8
    e'8
}
```

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



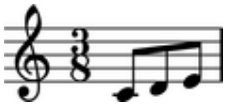
37.4 Working with dynamic measures

Dynamic measures adjust their time signatures on the fly as you add and remove music.

Create dynamic measures without a time signature:

```
>>> measure = measuretools.DynamicMeasure("c'8 d'8 e'8")
```

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

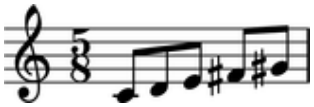


37.5 Adding music to dynamic measures

Add music to dynamic measures the same as to all containers:

```
>>> measure.extend([Note("fs'8"), Note("gs'8")])
```

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

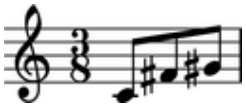


37.6 Removing music from dynamic measures

Remove music from dynamic measures the same as with other containers:

```
>>> del(measure[1:3])
```

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

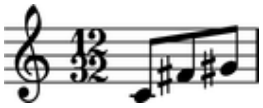


37.7 Setting the denominator of dynamic measures

You can set the denominator of dynamic measures to any integer power of 2:

```
>>> measure.denominator = 32
```

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



37.8 Suppressing the time signature of dynamic measures

You can temporarily suppress the time signature of dynamic measures:

```
>>> measure.suppress_time_signature = True
```

```
>>> f(measure)
{
    c'8
    fs'8
    gs'8
}
```

LilyPond will engrave the last active time signature.

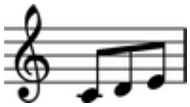
37.9 Working with anonymous measures

Anonymous determine their time signatures on the fly and then hide them at format time.

Create anonymous measures without a time signature:

```
>>> measure = measuretools.AnonymousMeasure("c'8 d'8 e'8")
```

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

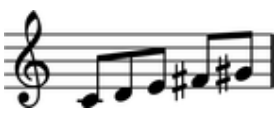


37.10 Adding music to anonymous measures

Add music to anonymous measures the same as to other containers:

```
>>> measure.extend([Note("fs'8"), Note("gs'8")])
```

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

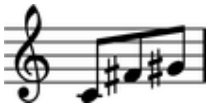


37.11 Removing music from anonymous measures

Remove music from anonymous measure the same as from other containers:

```
>>> del(measure[1:3])
```

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



CHAPTER

THIRTYEIGHT

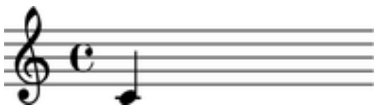
NOTES

38.1 Making notes from a string

You can make notes from string:

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

```
>>> show(note, docs=True)
```

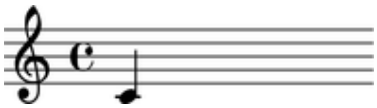


38.2 Making notes from chromatic pitch number and duration

You can also make notes from chromatic pitch number and duration:

```
>>> note = Note(0, Duration(1, 4))
```

```
>>> show(note, docs=True)
```



(You even use `Note("c'4")` to create notes with numbers alone.)

38.3 Getting the written pitch of notes

You can get the written pitch of notes:

```
>>> note.written_pitch
NamedChromaticPitch("c'")
```

38.4 Changing the written pitch of notes

And you can change the written pitch of notes:

```
>>> note.written_pitch = "cs'"
```

(You can use `note.written_pitch = 1` to change pitch with numbers, too.)

38.5 Getting the duration attributes of notes

Get the written duration of notes like this:

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

Which is usually the same as preprolated duration:

```
>>> note.preprolated_duration
Duration(1, 4)
```

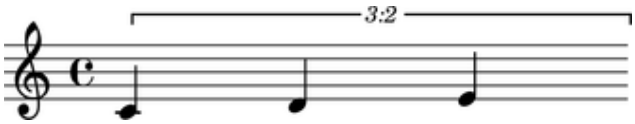
And prolated duration:

```
>>> note.prolated_duration
Duration(1, 4)
```

Except for notes inside a tuplet:

```
>>> tuplet = Tuplet(Fraction(2, 3), [Note("c'4"), Note("d'4"), Note("e'4")])
```

```
>>> show(tuplet, docs=True)
```




```
>>> note = tuplet[0]
```

Tupletted notes carry written duration:

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

Prolation:

```
>>> note.prolation
Multiplier(2, 3)
```

And prolated duration that is the product of the two:

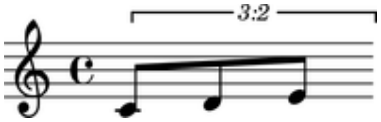
```
>>> note.prolated_duration
Duration(1, 6)
```

38.6 Changing the written duration of notes

You can change the written duration of notes:

```
>>> tuplet[0].written_duration = Duration(1, 8)
>>> tuplet[1].written_duration = Duration(1, 8)
>>> tuplet[2].written_duration = Duration(1, 8)
```

```
>>> show(tuplet, docs=True)
```



Other duration attributes are read-only.

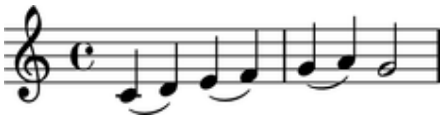
38.7 Overriding notes

The notes below are black with fixed thickness and predetermined spacing:

```
>>> staff = Staff("c'4 d'4 e'4 f'4 g'4 a'4 g'2")
>>> slur_1 = spannertools.SlurSpanner(staff[:2])
>>> slur_2 = spannertools.SlurSpanner(staff[2:4])
>>> slur_3 = spannertools.SlurSpanner(staff[4:6])
```

```
>>> f(staff)
\new Staff {
    c'4 (
    d'4 )
    e'4 (
    f'4 )
    g'4 (
    a'4 )
    g'2
}
```

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

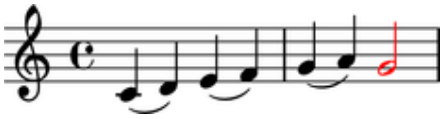


But you can override LilyPond grobs to change the look of notes, rests and chords:

```
>>> staff[-1].override.note_head.color = 'red'
>>> staff[-1].override.stem.color = 'red'
```

```
>>> f(staff)
\new Staff {
    c'4 (
    d'4 )
    e'4 (
    f'4 )
    g'4 (
    a'4 )
    \once \override NoteHead #'color = #red
    \once \override Stem #'color = #red
    g'2
}
```

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



38.8 Removing note overrides

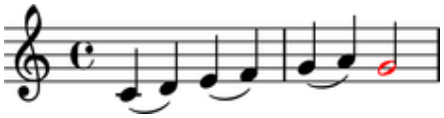
Delete grob overrides you no longer want:

```
>>> del(staff[-1].override.stem)
```

```
>>> f(staff)
\new Staff {
    c'4 (
    d'4 )
    e'4 (
    f'4 )
    g'4 (
    a'4 )
    \once \override NoteHead #'color = #red
```

```
    g'2
}
```

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



CHAPTER

THIRTYNINE

PITCHES

Named chromatic pitches are the everyday pitches attached to notes and chords:

```
>>> note = Note("cs''8")
```

```
>>> note.written_pitch
NamedChromaticPitch("cs''")
```

39.1 Creating pitches

Use pitch tools to create named chromatic pitches:

```
>>> named_chromatic_pitch = pitchtools.NamedChromaticPitch("cs''")
```

```
>>> named_chromatic_pitch
NamedChromaticPitch("cs''")
```

39.2 Inspecting the name of a pitch

Use `str()` to get the name of named chromatic pitches:

```
>>> str(named_chromatic_pitch)
"cs''"
```

39.3 Inspecting the octave of a pitch

Get the octave number of named chromatic pitches with `octave_number`:

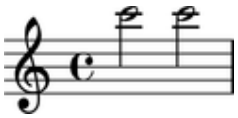
```
>>> named_chromatic_pitch.octave_number
5
```

39.4 Working with pitch deviation

Use deviation to model the fact that two pitches differ by a fraction of a semitone:

```
>>> note_1 = Note(24, (1, 2))
>>> note_2 = Note(24, (1, 2))
>>> staff = Staff([note_1, note_2])
```

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



```
>>> note_2.written_pitch = pitchtools.NamedChromaticPitch(24, deviation = -31)
```

The pitch of the the first note is greater than the pitch of the second:

```
>>> note_1.written_pitch > note_2.written_pitch
False
```

Use markup to include indications of pitch deviation in your score:

```
>>> markuptools.Markup(note_2.written_pitch.deviation_in_cents, Up)(note_2)
Markup(None, direction=Up)(c'''2)
```

39.5 Sorting pitches

Named chromatic pitches sort by octave, diatonic pitch-class and accidental, in that order:

```
>>> pitchtools.NamedChromaticPitch('es') < pitchtools.NamedChromaticPitch('ff')
True
```

39.6 Comparing pitches

Compare named chromatic pitches to each other:

```
>>> named_chromatic_pitch_1 = pitchtools.NamedChromaticPitch("c''")
>>> named_chromatic_pitch_2 = pitchtools.NamedChromaticPitch("d''")
```

```
>>> named_chromatic_pitch_1 == named_chromatic_pitch_2
False
```

```
>>> named_chromatic_pitch_1 != named_chromatic_pitch_2
True
```

```
>>> named_chromatic_pitch_1 > named_chromatic_pitch_2
False
```

```
>>> named_chromatic_pitch_1 < named_chromatic_pitch_2
True
```

```
>>> named_chromatic_pitch_1 >= named_chromatic_pitch_2
False
```

```
>>> named_chromatic_pitch_1 <= named_chromatic_pitch_2
True
```

39.7 Converting one type of pitch to another

Convert any named chromatic pitch to a named diatonic pitch:

```
>>> named_chromatic_pitch.named_diatonic_pitch
NamedDiatonicPitch("c''")
```

To a numbered chromatic pitch:

```
>>> named_chromatic_pitch.numbered_chromatic_pitch
NumberedChromaticPitch(13)
```

Or to a numbered diatonic pitch:

```
>>> named_chromatic_pitch.numbered_diatonic_pitch
NumberedDiatonicPitch(7)
```

39.8 Converting pitches to pitch-classes

Convert any named chromatic pitch to a named chromatic pitch-class:

```
>>> named_chromatic_pitch.named_chromatic_pitch_class
NamedChromaticPitchClass('cs')
```

To a named diatonic pitch-class:

```
>>> named_chromatic_pitch.named_diatonic_pitch_class
NamedDiatonicPitchClass('c')
```

To a numbered chromatic pitch-class:

```
>>> named_chromatic_pitch.numbered_chromatic_pitch_class
NumberedChromaticPitchClass(1)
```

Or to a numbered diatonic pitch-class:

```
>>> named_chromatic_pitch.numbered_diatonic_pitch_class
NumberedDiatonicPitchClass(0)
```

39.9 Copying pitches

Use `copy.copy()` to copy named chromatic pitches:

```
>>> import copy
```

```
>>> copy.copy(named_chromatic_pitch)
NamedChromaticPitch("cs''")
```

Or use `copy.deepcopy()` to do the same thing.

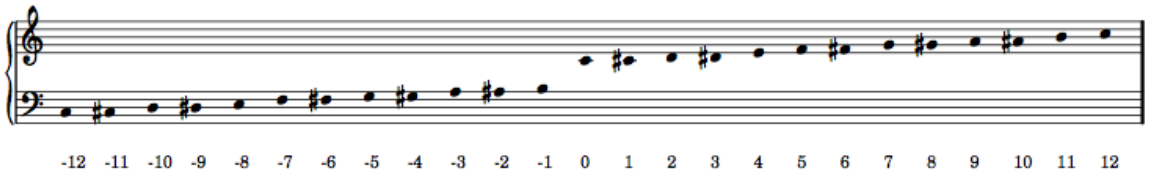
39.10 Accidental abbreviations

Abjad abbreviates accidentals according to the LilyPond `english.ly` module:

accidental name	abbreviation
quarter sharp	'qs'
quarter flat	'qf'
sharp	's'
flat	'f'
three-quarters sharp	'tqs'
three-quarters flat	'tqf'
double sharp	'ss'
double flat	'ff'

39.11 Chromatic pitch numbers

Abjad numbers chromatic pitches by semitone with middle C set equal to 0:

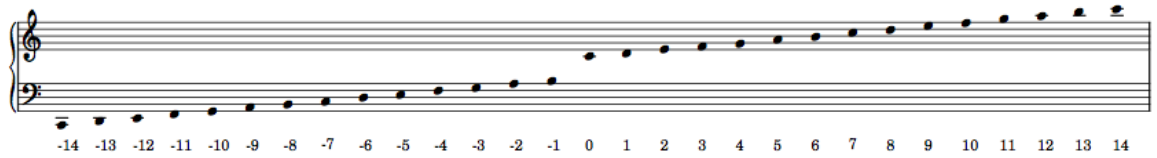


The code to generate this table is as follows:

```
score, treble_staff, bass_staff = scoretools.make_empty_piano_score()
duration = Fraction(1, 32)

treble = measuretools.AnonymousMeasure([])
bass = measuretools.AnonymousMeasure([])

treble_staff.append(treble)
bass_staff.append(bass)

pitches = range(-12, 12 + 1)

pitchtools.set_default_accidental_spelling('sharps')

for i in pitches:
    note = Note(i, duration)
    rest = Rest(duration)
    clef = pitchtools.suggest_clef_for_named_chromatic_pitches([note.pitch])
    if clef == contexttools.ClefMark('treble'):
        treble.append(note)
        bass.append(rest)
    else:
        treble.append(rest)
        bass.append(note)
    diatonic_pitch_number = str(note.pitch.numbered_chromatic_pitch)
    markuptools.Markup(diatonic_pitch_number, Down)(bass[-1])

score.override.rest.transparent = True
score.override.stem.stencil = False
```

39.12 Diatonic pitch numbers

Abjad numbers diatonic pitches by staff space with middle C set equal to 0:



The code to generate this table is as follows:

```
score, treble_staff, bass_staff = scoretools.make_empty_piano_score()
duration = Fraction(1, 32)

treble = measuretools.AnonymousMeasure([])
bass = measuretools.AnonymousMeasure([])

treble_staff.append(treble)
bass_staff.append(bass)

pitches = []
diatonic_pitches = [0, 2, 4, 5, 7, 9, 11]

pitches.extend([-24 + x for x in diatonic_pitches])
pitches.extend([-12 + x for x in diatonic_pitches])
pitches.extend([0 + x for x in diatonic_pitches])
```

```
pitches.extend([12 + x for x in diatonic_pitches])
pitches.append(24)
pitchtools.set_default_accidental_spelling('sharps')

for i in pitches:
    note = Note(i, duration)
    rest = Rest(duration)
    clef = pitchtools.suggest_clef_for_named_chromatic_pitches([note.pitch])
    if clef == contexttools.ClefMark('treble'):
        treble.append(note)
        bass.append(rest)
    else:
        treble.append(rest)
        bass.append(note)
    diatonic_pitch_number = abs(note.pitch.numbered_diatonic_pitch)
    markuptools.Markup(diatonic_pitch_number, Down)(bass[-1])

score.override.rest.transparent = True
score.override.stem.stencil = False
```

39.13 Octave designation

Abjad designates octaves with both numbers and ticks:

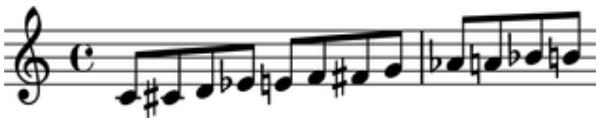
Octave notation	Tick notation
C7	c''''
C6	c'''
C5	c''
C4	c'
C3	c
C2	c,
C1	c,,

39.14 Accidental spelling

Abjad chooses between enharmonic spellings at pitch-initialization according to the following table:

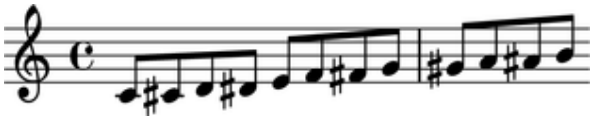
Chromatic pitch-class number	Chromatic pitch-class name (default)
0	C
1	C#
2	D
3	Eb
4	E
5	F
6	F#
7	G
8	Gb
9	A
10	Bb
11	B

```
>>> staff = Staff([Note(n, (1, 8)) for n in range(12)])
>>> show(staff)
```



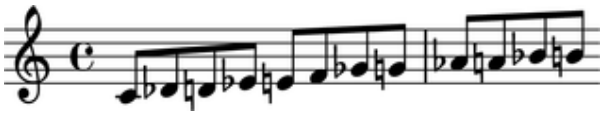
Use pitch tools to respell with sharps:

```
>>> pitchtools.respell_named_chromatic_pitches_in_expr_with_sharps(staff)
>>> show(staff)
```



Or flats:

```
>>> pitchtools.respell_named_chromatic_pitches_in_expr_with_flats(staff)
>>> show(staff)
```



CHAPTER

FORTY

WORKING WITH LISTS OF NUMBERS

Python provides a built-in `list` class that you can use to carry around almost anything. The examples here show how to create a list of numbers and then do things with the numbers in the list.

Create a list with square brackets.

```
>>> my_list = [23, 7, 10, 18, 13, 20, 3, 2, 18, 9, 14, 3]
>>> my_list
[23, 7, 10, 18, 13, 20, 3, 2, 18, 9, 14, 3]
```

Use `len()` to find the number of elements in any list.

```
>>> len(my_list)
12
```

Use `append()` to add one element to a list.

```
>>> my_list.append(5)
>>> my_list
[23, 7, 10, 18, 13, 20, 3, 2, 18, 9, 14, 3, 5]
```

Use `extend()` to extend one list with the contents of another.

```
>>> my_other_list = [19, 11, 4, 10, 12]
>>> my_list.extend(my_other_list)
>>> my_list
[23, 7, 10, 18, 13, 20, 3, 2, 18, 9, 14, 3, 5, 19, 11, 4, 10, 12]
```

Use `reverse()` to reverse the elements in a list.

```
>>> my_list.reverse()
>>> my_list
[12, 10, 4, 11, 19, 5, 3, 14, 9, 18, 2, 3, 20, 13, 18, 10, 7, 23]
```

You can return a single value from a list with a numeric index.

```
>>> my_list[0]
12
>>> my_list[1]
10
>>> my_list[2]
4
```

You can return many values from a list with slice notation.

```
>>> my_list[:4]
[12, 10, 4, 11]
```

More information on these and all other operations defined on the built-in Python `list` is available in the Python tutorial.

CHAPTER

FORTYONE

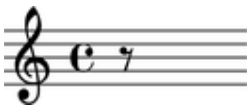
RESTS

41.1 Making rests from strings

You can make rests from a string:

```
>>> rest = Rest('r8')
```

```
>>> show(rest, docs=True)
```



41.2 Making rests from durations

You can also make rests from a duration:

```
>>> rest = Rest(Duration(1, 4))
```

```
>>> show(rest, docs=True)
```

(You can even use `Rest((1, 8))` to make rests from a duration pair.)

41.3 Getting the duration attributes of rests

Get the written duration of rests like this:

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

Which is usually the same as preprolated duration:

```
>>> rest.preprolated_duration
Duration(1, 4)
```

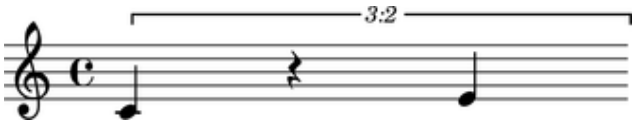
And prolated duration:

```
>>> rest.prolated_duration
Duration(1, 4)
```

Except for rests inside a tuplet:

```
>>> tuplet = Tuplet(Fraction(2, 3), [Note("c'4"), Rest('r4'), Note("e'4")])
```

```
>>> show(tuplet, docs=True)
```




```
>>> rest = tuplet[1]
```

Tupletted rests carry written duration:

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

Prolation:

```
>>> rest.prolation
Multiplier(2, 3)
```

And prolated duration that is the product of the two:

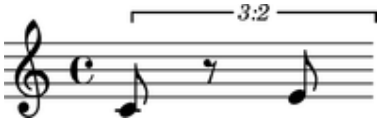
```
>>> rest.prolated_duration
Duration(1, 6)
```

41.4 Changing the written duration of rests

You can change the written duration of notes and rests:

```
>>> tuplet[0].written_duration = Duration(1, 8)
>>> tuplet[1].written_duration = Duration(1, 8)
>>> tuplet[2].written_duration = Duration(1, 8)
```

```
>>> show(tuplet, docs=True)
```



Other duration attributes are read-only.

CHAPTER

FORTYTWO

SCORES

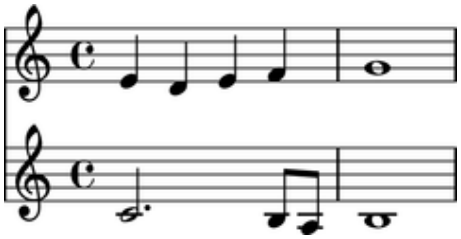
42.1 Creating scores

Create a score like this:

```
>>> treble_staff_1 = Staff("e'4 d'4 e'4 f'4 g'1")
>>> treble_staff_2 = Staff("c'2. b8 a8 b1")
```

```
>>> score = Score([treble_staff_1, treble_staff_2])
```

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



42.2 Inspecting score music

Return score components with `music`:

```
>>> score.music
(Staff{5}, Staff{4})
```

42.3 Inspecting score length

Get score length with `len()`:

```
>>> len(score)
2
```

42.4 Inspecting score duration

Score contents duration is equal to the duration of the longest component in score:

```
>>> score.contents_duration
Duration(2, 1)
```

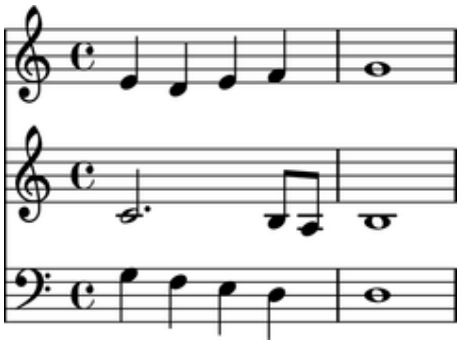
42.5 Adding one component to the bottom of a score

Add one component to the bottom of a score with `append`:

```
>>> bass_staff = Staff("g4 f4 e4 d4 d1")
>>> contexttools.ClefMark('bass')(bass_staff)
ClefMark('bass')(Staff{5})
```

```
>>> score.append(bass_staff)
```

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



42.6 Finding the index of a score component

Find the index of a score component with `index`:

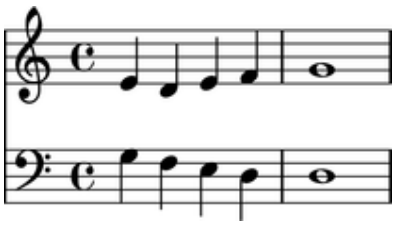
```
>>> score.index(treble_staff_1)
0
```

42.7 Removing a score component by index

Use `pop` to remove a score component by index:

```
>>> score.pop(1)
Staff{4}
```

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

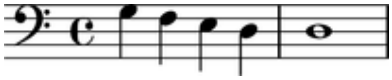


42.8 Removing a score component by reference

Remove a score component by reference with `remove`:

```
>>> score.remove(treble_staff_1)
```

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



42.9 Testing score containment

Use `in` to find out whether a score contains a given component:

```
>>> treble_staff_1 in score
False
```

```
>>> treble_staff_2 in score
False
```

```
>>> bass_staff in score
True
```

42.10 Naming scores

You can name Abjad scores:

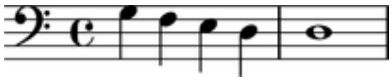
```
>>> score.name = 'Example Score'
```

Score names appear in LilyPond input:

```
>>> f(score)
\context Score = "Example Score" <<
    \new Staff {
        \clef "bass"
        g4
        f4
        e4
        d4
        d1
    }
>>
```

But do not appear in notational output:

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



CHAPTER

FORTYTHREE

SPANNERS

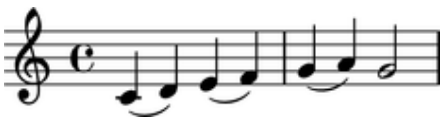
43.1 Overriding spanners

The symbols below are black with fixed thickness and predetermined spacing:

```
>>> staff = Staff("c'4 d'4 e'4 f'4 g'4 a'4 g'2")
>>> slur_1 = spannertools.SlurSpanner(staff[:2])
>>> slur_2 = spannertools.SlurSpanner(staff[2:4])
>>> slur_3 = spannertools.SlurSpanner(staff[4:6])
```

```
>>> f(staff)
\new Staff {
    c'4 (
    d'4 )
    e'4 (
    f'4 )
    g'4 (
    a'4 )
    g'2
}
```

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

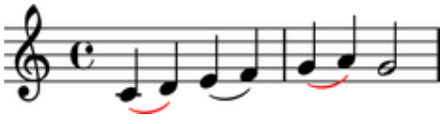


But you can override LilyPond grobs to change the look of spanners:

```
>>> slur_1.override.slur.color = 'red'
>>> slur_3.override.slur.color = 'red'
```

```
>>> f(staff)
\new Staff {
    \override Slur #'color = #red
    c'4 (
    d'4 )
    \revert Slur #'color
    e'4 (
    f'4 )
    \override Slur #'color = #red
    g'4 (
    a'4 )
    \revert Slur #'color
    g'2
}
```

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



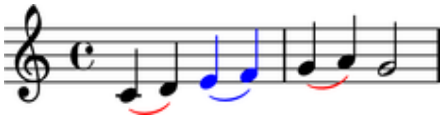
43.2 Overriding the components to which spanners attach

You can override LilyPond grobs to change spanners' contents:

```
>>> slur_2.override.slur.color = 'blue'
>>> slur_2.override.note_head.color = 'blue'
>>> slur_2.override.stem.color = 'blue'
```

```
>>> f(staff)
\new Staff {
    \override Slur #'color = #red
    c'4 (
    d'4 )
    \revert Slur #'color
    \override NoteHead #'color = #blue
    \override Slur #'color = #blue
    \override Stem #'color = #blue
    e'4 (
    f'4 )
    \revert NoteHead #'color
    \revert Slur #'color
    \revert Stem #'color
    \override Slur #'color = #red
    g'4 (
    a'4 )
    \revert Slur #'color
    g'2
}
```

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



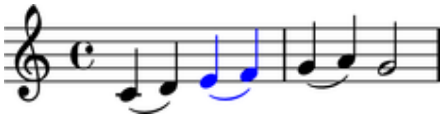
43.3 Removing spanner overrides

Delete grob overrides you no longer want:

```
>>> del(slur_1.override.slur)
>>> del(slur_3.override.slur)
```

```
>>> f(staff)
\new Staff {
    c'4 (
    d'4 )
    \override NoteHead #'color = #blue
    \override Slur #'color = #blue
    \override Stem #'color = #blue
    e'4 (
    f'4 )
    \revert NoteHead #'color
    \revert Slur #'color
    \revert Stem #'color
    g'4 (
    a'4 )
    g'2
}
```

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



CHAPTER

FORTYFOUR

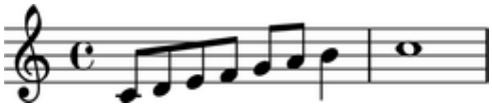
STAVES

44.1 Creating staves

Create staves like this:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'4 c''1")
```

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



44.2 Inspecting staff music

Return staff components with `music`:

```
>>> staff.music
(Note("c'8"), Note("d'8"), Note("e'8"), Note("f'8"), Note("g'8"), Note("a'8"), Note("b'4"), Note("c''1"))
```

44.3 Inspecting staff length

Get staff length with `len()`:

```
>>> len(staff)
8
```

44.4 Inspecting staff duration

Staff contents durations equals the sum of staff components' duration:

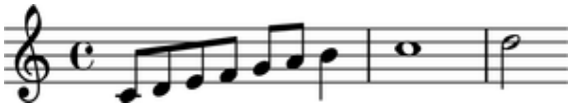
```
>>> staff.contents_duration
Duration(2, 1)
```

44.5 Adding one component to the end of a staff

Add one component to the end of a staff with `append`:

```
>>> staff.append(Note("d''2"))
```

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

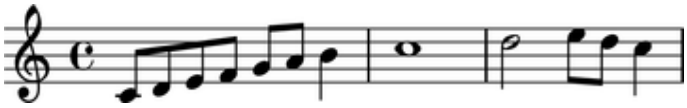


44.6 Adding many components to the end of a staff

Add many components to the end of a staff with `extend`:

```
>>> notes = [Note("e''8"), Note("d''8"), Note("c''4")]
>>> staff.extend(notes)
```

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



44.7 Finding the index of a staff component

Find staff component index with `index`:

```
>>> notes[0]
Note("e''8")
```

```
>>> staff.index(notes[0])
9
```

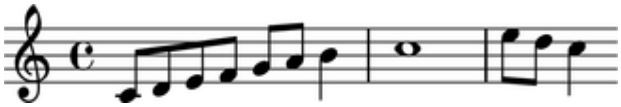
44.8 Removing a staff component by index

Use `pop` to remove a staff component by index:

```
>>> staff[8]
Note("d''2")
```

```
>>> staff.pop(8)
Note("d''2")
```

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

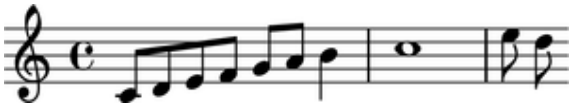


44.9 Removing a staff component by reference

Remove staff components by reference with `remove`:

```
>>> staff.remove(staff[-1])
```

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



44.10 Naming staves

You can name Abjad staves:

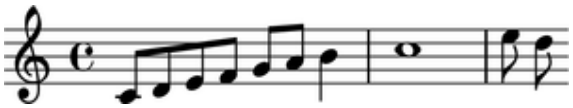
```
>>> staff.name = 'Example Staff'
```

Staff names appear in LilyPond input:

```
>>> f(staff)
\context Staff = "Example Staff" {
    c'8
    d'8
    e'8
    f'8
    g'8
    a'8
    b'4
    c''1
    e''8
    d''8
}
```

But not in notational output:

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



44.11 Forcing context

Staff context equals `'Staff'` by default:

```
>>> staff.context_name
'Staff'
```

You can force staff context:

```
>>> staff.context_name = 'CustomUserStaff'
```

```
>>> staff.context_name
'CustomUserStaff'
```

```
>>> f(staff)
\context CustomUserStaff = "Example Staff" {
    c'8
    d'8
    e'8
    f'8
    g'8
    a'8
    b'4
    c''1
    e''8
    d''8
}
```

Force context when you have defined a new LilyPond context.

CHAPTER

FORTYFIVE

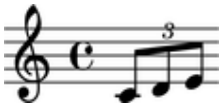
TUPLETS

45.1 Making a tuplet from a LilyPond input string

You can make an Abjad tuplet from a multiplier and a LilyPond input string:

```
>>> tuplet = Tuplet(Fraction(2, 3), "c'8 d'8 e'8")
```

```
>>> show(tuplet)
```



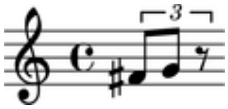
45.2 Making a tuplet from a list of other Abjad components

You can also make a tuplet from a multiplier and a list of other Abjad components:

```
>>> leaves = [Note("fs'8"), Note("g'8"), Rest('r8')]
```

```
>>> tuplet = Tuplet(Fraction(2, 3), leaves)
```

```
>>> show(tuplet)
```



45.3 Understanding the interpreter display of a tuplet

The interprer display of an Abjad tuplet contains three parts:

```
>>> tuplet
Tuplet(2/3, [fs'8, g'8, r8])
```

`Tuplet` tells you the tuplet's class.

`2/3` tells you the tuplet's multiplier.

The list `[fs'8, g'8, r8]` shows the top-level components the tuplet contains.

45.4 Understanding the string representation of a tuplet

The string representation of a tuplet contains four parts:

```
>>> print tuplet
{* 3:2 fs'8, g'8, r8 *}
```

Curly braces `{` and `}` indicate that the tuplet's music is interpreted sequentially instead of in parallel.

The asterisks `*` denote a fixed-multiplier tuplet.

`3:2` tells you the tuplet's ratio.

The remaining arguments show the top-level components of tuplet.

45.5 Inspecting the LilyPond format of a tuplet

Get the LilyPond input format of any Abjad object with `format`:

```
>>> tuplet.lilypond_format
"\\times 2/3 {\n\tfs'8\n\tg'8\n\tr8\n}"
```

Use `f()` as a short-cut to print the LilyPond format of any Abjad object:

```
>>> f(tuplet)
\times 2/3 {
    fs'8
    g'8
    r8
}
```

45.6 Inspecting the music in a tuplet

Get the music in any Abjad container with `music`:

```
>>> tuplet.music
(Note("fs'8"), Note("g'8"), Rest('r8'))
```

Abjad returns a read-only tuple of components.

45.7 Inspecting a tuplet's leaves

Get the leaves in any Abjad container with `leaves`:

```
>>> tuplet.leaves
(Note("fs'8"), Note("g'8"), Rest('r8'))
```

Abjad returns a read-only tuple of leaves.

45.8 Getting the length of a tuplet

Get the length of any Abjad container with `len()`:

```
>>> len(tuplet)
3
```

The length of every Abjad container is defined equal to the number of top-level components present in the container.

45.9 Getting the duration attributes of a tuplet

You set the multiplier of a tuplet at initialization:

```
>>> tuplet.multiplier
Multiplier(2, 3)
```

The contents durations of a tuplet equals the sum of written durations of the components in the tuplet:

```
>>> tuplet.contents_duration
Duration(3, 8)
```

The multiplied duration of a tuplet equals the product of the tuplet's multiplier and the tuplet's contents duration:

```
>>> tuplet.multiplied_duration
Duration(1, 4)
```

45.10 Understanding rhythmic augmentation and diminution

A tuplet with a multiplier less than `1` constitutes a type of rhythmic diminution:

```
>>> tuplet.multiplier
Multiplier(2, 3)
```

```
>>> tuplet.is_diminution
True
```

A tuplet with a multiplier greater than `1` is a type of rhythmic augmentation:

```
>>> tuplet.is_augmentation
False
```

45.11 Understanding binary and nonbinary tuplets

A tuplet is considered binary if the numerator of the tuplet multiplier is an integer power of `2`:

```
>>> tuplet.multiplier
Multiplier(2, 3)
```

```
>>> tuplet.has_power_of_two_denominator
True
```

Other tuplets are nonbinary:

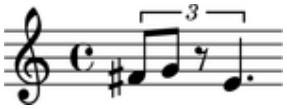
```
>>> tuplet.has_non_power_of_two_denominator
False
```

45.12 Adding one component to the end of a tuplet

Add one component to the end of a tuplet with `append`:

```
>>> tuplet.append(Note("e'4."))
```

```
>>> show(tuplet)
```

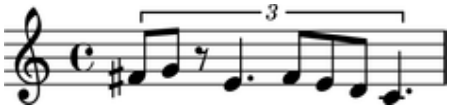


45.13 Adding many components to the end of a tuplet

Add many components to the end of a tuplet with `extend`:

```
>>> notes = [Note("fs'8"), Note("e'8"), Note("d'8"), Note("c'4.")]
>>> tuplet.extend(notes)
```

```
>>> show(tuplet)
```



45.14 Finding the index of a component in a tuplet

Find the index of a component in a tuplet with `index()`:

```
>>> notes[1]
Note("e'8")
```

```
>>> tuplet.index(notes[1])
5
```

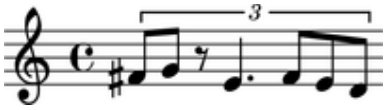
45.15 Removing a tuplet component by index

Use `pop()` to remove a tuplet component by index:

```
>>> tuplet[7]
Note("c'4.")
```

```
>>> tuplet.pop(7)
Note("c'4.")
```

```
>>> show(tuplet)
```

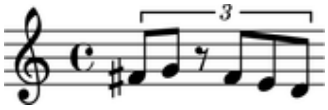


45.16 Removing a tuplet component by reference

Remove tuplet components by reference with `remove()`:

```
>>> tuplet.remove(tuplet[3])
```

```
>>> show(tuplet)
```



45.17 Overriding attributes of the LilyPond tuplet number grob

Override attributes of the LilyPond tuplet number grob like this:

```
>>> tuplet.override.tuplet_number.text = schemetools.Scheme('tuplet-number::calc-fraction-text')
>>> tuplet.override.tuplet_number.color = 'red'
```

```
>>> f(tuplet)
\override TupletNumber #'color = #red
\override TupletNumber #'text = #tuplet-number::calc-fraction-text
\times 2/3 {
    fs'8
    g'8
    r8
    fs'8
    e'8
    d'8
}
\revert TupletNumber #'color
\revert TupletNumber #'text
```

```
>>> show(tuplet)
```

See the LilyPond docs for lists of grob attributes available.

45.18 Overriding attributes of the LilyPond tuplet bracket grob

Override attributes of the LilyPond tuplet bracket grob like this:

```
>>> tuplet.override.tuplet_bracket.color = 'red'
```

```
>>> f(tuplet)
\override TupletBracket #'color = #red
\override TupletNumber #'color = #red
\override TupletNumber #'text = #tuplet-number::calc-fraction-text
\times 2/3 {
    fs'8
    g'8
    r8
    fs'8
    e'8
    d'8
}
\revert TupletBracket #'color
\revert TupletNumber #'color
\revert TupletNumber #'text
```

```
>>> show(tuplet)
```

See the LilyPond docs for lists of grob attributes available.

CHAPTER

FORTYSIX

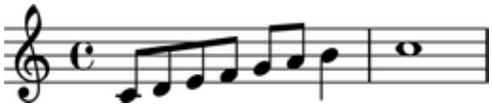
VOICES

46.1 Making a voice from a LilyPond input string

You can make an Abjad voice from a LilyPond input string:

```
>>> voice = Voice("c'8 d'8 e'8 f'8 g'8 a'8 b'4 c''1")
```

```
>>> show(voice)
```



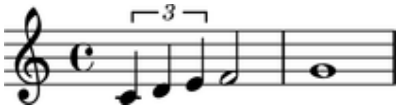
46.2 Making a voice from a list of other Abjad components

You can also make a voice from a list of other Abjad components:

```
>>> components = [Tuplet(Fraction(2, 3), "c'4 d'4 e'4"), Note("f'2"), Note("g'1")]
```

```
>>> voice = Voice(components)
```

```
>>> show(voice)
```



46.3 Understanding the `repr` of a voice

The `repr` of an Abjad voice contains three parts:

```
>>> voice
Voice{3}
```

`Voice` tells you the voice's class.

`3` tells you the voice's length (which is the number of top-level components the voice contains).

Curly braces `{` and `}` tell you that the music inside the voice is interpreted sequentially rather than in parallel.

46.4 Inspecting the LilyPond format of a voice

Get the LilyPond input format of any Abjad object with `format`:

```
>>> voice.lilypond_format
"\\new Voice {\n\t\\times 2/3 {\n\t\tc'4\n\t\td'4\n\t\te'4\n\t}\n\tf'2\n\tg'1\n}"
```

Use `f()` as a short-cut to print the LilyPond format of any Abjad object:

```
>>> f(voice)
\new Voice {
    \times 2/3 {
        c'4
        d'4
        e'4
    }
    f'2
    g'1
}
```

46.5 Inspecting the music in a voice

Get voice components with `music`:

```
>>> voice.music
(Tuplet(2/3, [c'4, d'4, e'4]), Note("f'2"), Note("g'1"))
```

Abjad returns a read-only tuple of components.

46.6 Inspecting a voice's leaves

Get the leaves in a voice with `leaves`:

```
>>> voice.leaves
(Note("c'4"), Note("d'4"), Note("e'4"), Note("f'2"), Note("g'1"))
```

Abjad returns a read-only tuple of leaves.

46.7 Getting the length of a voice

Get voice length with `len()`:

```
>>> len(voice)
3
```

The length of a voice is defined equal to the number of top-level components the voice contains.

46.8 Getting the duration attributes of a voice

The contents durations of a voice equals the sum of durations of the components in the voice:

```
>>> voice.contents_duration
Duration(2, 1)
```

The preprolated duration of a voice is usually equal to the voice's contents duration:

```
>>> voice.preprolated_duration
Duration(2, 1)
```

The prolated duration of a voice is usually equal to the voice's contents duration, too:

```
>>> voice.preprolated_duration
Duration(2, 1)
```

Only when you nest a very small voice inside a tuplet will the prolated and preprolated duration of a voice differ.

Voices that are not nested inside a tuplet carry a prolation of `1`:

```
>>> voice.prolation
Fraction(1, 1)
```

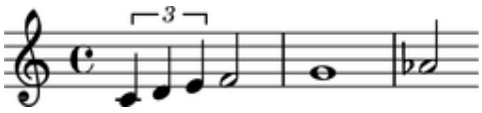
All voice duration attributes are read-only.

46.9 Adding one component to the end of a voice

Add one component to the end of a voice with `append`:

```
>>> voice.append(Note("af'2"))
```

```
>>> show(voice)
```

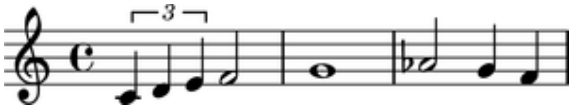


46.10 Adding many components to the end of a voice

Add many components to the end of a voice with `extend`:

```
>>> notes = [Note("g'4"), Note("f'4")]
>>> voice.extend(notes)
```

```
>>> show(voice)
```



46.11 Finding the index of a component in a voice

Find the index of a component in a voice with `index()`:

```
>>> notes[0]
Note("g'4")
```

```
>>> voice.index(notes[0])
4
```

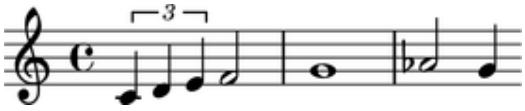
46.12 Removing a voice component by index

Use `pop()` to remove a voice component by index:

```
>>> voice[5]
Note("f'4")
```

```
>>> voice.pop(5)
Note("f'4")
```

```
>>> show(voice)
```

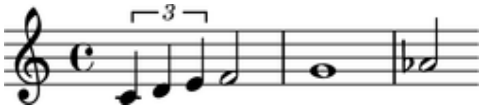


46.13 Removing a voice component by reference

Remove voice components by reference with `remove()`:

```
>>> voice.remove(voice[-1])
```

```
>>> show(voice)
```



46.14 Naming voices

You can name Abjad voices:

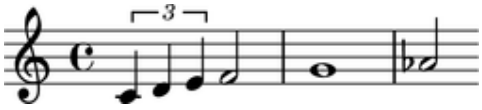
```
>>> voice.name = 'Upper Voice'
```

Voice names appear in LilyPond input:

```
>>> f(voice)
\context Voice = "Upper Voice" {
    \times 2/3 {
        c'4
        d'4
        e'4
    }
    f'2
    g'1
    af'2
}
```

But not in notational output:

```
>>> show(voice)
```



46.15 Changing the context of a voice

The context of a voice is set to `'Voice'` by default:

```
>>> voice.context_name
'Voice'
```

But you can change the context of a voice if you want:

```
>>> voice.context_name = 'SpeciallyDefinedVoice'
```

```
>>> voice.context_name
'SpeciallyDefinedVoice'
```

```
>>> f(voice)
\context SpeciallyDefinedVoice = "Upper Voice" {
    \times 2/3 {
        c'4
        d'4
        e'4
    }
    f'2
    g'1
    af'2
}
```

Change the context of a voice when you have defined a new LilyPond context based on a LilyPond voice.

Part VI

Developer documentation

CHAPTER

FORTYSEVEN

CODEBASE

47.1 How the Abjad codebase is laid out

The Abjad codebase comprises a small number of top-level directories:

```
abjad$ ls -x -F --width 80
ls: illegal option -- -
usage: ls [-ABCFGHLOPRSTUWabcdefghiklmnopqrstuwx1] [file ...]
```

Of these, is is in the `tools` directory that the bulk of the musical reasoning implemented in Abjad resides:

```
abjad$ ls tools/ -x -F --width 80
tools/:
__init__.py
__init__.pyc
abctools
abjadbooktools
beamtools
chordtools
componenttools
configurationtools
containertools
contexttools
datastructuretools
decoratortools
developerscripttools
documentationtools
durationtools
exceptiontools
formattools
gracetools
importtools
instrumenttools
introspectiontools
iotools
iterationtools
labeltools
layouttools
leaftools
lilypondfiletools
lilypondparsertools
lilypondproxytools
marktools
markuptools
mathtools
measuretools
notetools
offsettools
pitcharraytools
pitchtools
quantizationtools
resttools
rhythmmakertools
rhythmtreetools
schemetools
```

```
scoretemplatetools
scoretools
selectiontools
sequencetools
sievetools
skiptools
spannertools
stafftools
stringtools
tempotools
testtools
tietools
timeintervaltools
timerelationtools
timesignaturetools
timespantools
tonalitytools
tuplettools
verticalitytools
voicetools
wellformednesstools
ls: --width: No such file or directory
ls: -F: No such file or directory
ls: -x: No such file or directory
ls: 80: No such file or directory
```

The remaining sections of this chapter cover the topics necessary to familiarize developers coming to the project for the first time.

47.2 Removing prebuilt versions of Abjad before you check out

If you'd like to be at the cutting edge of the Abjad development then you should check out from Google Code and tell Python and your operating system about Abjad. You can do this by following the steps below.

But before you do this you should realize that there are two ways to get Abjad up and running on your computer. The first way is by downloading a compressed version of Abjad from the Python Package Index. You probably did this when you first discovered Abjad and started to use the system. The second way is by following the steps below to check out a copy of the most recent version of the Abjad repository hosted on Google Code. If you already have a version of Abjad running on your computer but you haven't yet followed the steps below to check out from Google Code, then you probably downloaded a compressed version of Abjad from the Python Package Index.

Before you check out from Google Code you should remove all prebuilt versions of Abjad from your machine.

The reason you need to do this is that having both a prebuilt version of Abjad and a Subversion-managed version of Abjad on your machine can confuse your operating system and lead to weird results when you try to start Abjad.

You remove prebuilt versions of Abjad resident on your computer by finding your site packages directory and removing the so-called Abjad 'egg' that Python has installed there. After you remove the Abjad egg from your site packages directory you will also need to remove the `abj`, `abjad` and `abjad-book` scripts from `/usr/local/bin` or from the directory that is equivalent to `/usr/local/bin` under your opearting system.

First note the version of Python you're currently running:

```
abjad$ python --version
Python 2.7.3
```

This is important because you may have more than one version of Python installed on your machine. (Which tends especially to be the case if you're running a Apple's OS X.)

Then note that the site packages directory is a part of your filesystem into which Python installs third-party Python packages like Abjad. The location of the site packages directory varies from one operating system to the next and you may have to Google to find the exact location of the site packages directory on your machine. Under OS

X you can check `/Library/Python/2.x/site-packages/`. Under Linux the site packages directory is usually `/usr/lib/python2.x/site-packages`.

Once you've found your site packages directory you can list its contents to see if Python has installed an Abjad egg in it:

```
site-packages$ ls
Abjad-2.0-py2.6.egg         Sphinx-1.0.7-py2.6.egg      py-1.3.4-py2.6.egg
Jinja2-2.5-py2.6.egg        docutils-0.7-py2.6.egg      py-1.4.0-py2.6.egg
Pygments-1.3.1-py2.6.egg    easy-install.pth            py-1.4.4-py2.6.egg
README                      guppy                       pytest-2.0.0-py2.6.egg
Sphinx-1.0.1-py2.6.egg      guppy-0.1.9-py2.6.egg-info  pytest-2.1.0-py2.6.egg
Sphinx-1.0.4-py2.6.egg      py-1.3.1-py2.6.egg
```

Remove any Abjad eggs Python has installed in your site packages directory.

After you've done this you should check `/usr/local/bin` or equivalent to see if the `abj`, `abjad` or `abjad-book` scripts are installed there:

```
bin$ ls
abj      abjad    abjad-book
```

Remove any of the three scripts you find installed there so that you can use the new versions of the scripts you will download from Google Code instead:

```
bin$ sudo rm abj*
```

Now proceed to the steps below to check out from Google Code.

47.3 Installing the development version

Follow the steps listed above to remove prebuilt versions of Abjad from your machine. Then follow the steps below to check out from Google Code.

1. Make sure Subversion is installed on your machine:

   ```
   svn --version
   ```

   If Subversion responds then it is already installed. Otherwise visit the Subversion website.

2. Check out a copy of the main line of the Abjad codebase:

   ```
   svn checkout http://abjad.googlecode.com/svn/abjad/trunk abjad-trunk
   ```

3. Add the abjad trunk directory to your your `PYTHONPATH` environment variable:

   ```
   export PYTHONPATH="/path/to/abjad-trunk:"$PYTHONPATH
   ```

4. Alternatively you may symlink your Python site packages directory to the abjad trunk directory:

   ```
   ln -s /path/to/abjad-trunk /path/to/site-package/abjad
   ```

5. Finally, add `abjad-trunk/scr/` to your `PATH` environment variable:

   ```
   export PATH="/path/to/abjad-trunk/scr:"$PATH
   ```

   You will then be able to run Abjad with the `abjad` command.

You now have a copy of the main line of the most recent version of the Abjad repository checked out to your machine.

CHAPTER

FORTYEIGHT

DOCS

The reST-based sources for the Abjad documentation are included in their entirety in every installation of Abjad. You may add to and edit these reST-based sources as soon as you install Abjad. However, to build human-readable HTML or PDF versions of the docs you will first need to download and install Sphinx.

The remaining sections of this chapter describe how the Abjad docs are laid out and how to build the docs with Sphinx.

48.1 How the Abjad docs are laid out

The source files for the Abjad docs are included in the `docs` directory of every Abjad install. The `docs` directory contains everything required to build HTML, PDF and other versions of the Abjad docs.

```
abjad$ ls docs/
Makefile    _templates  chapters    index.rst  scr
_static     _themes     conf.py     make.bat
```

The bulk of the Abjad docs live in `docs/chapters`. The chapter directories mirror the main sections on Abjad documentation. What you'll find as you inspect the chapter directories are a collection of `.rst` files organized into groups. The `.rst` extension identifies files written in restructured text.

One example:

```
abjad$ ls docs/chapters/appendices/glossary
index.rst
```

48.2 Installing Sphinx

Sphinx is the automated documentation system used by Python, Abjad and other projects implemented in Python. Because Sphinx is not included in the Python standard library you will probably need to download and install it.

First check to see if Sphinx is already installed on your machine.

```
$ sphinx-build --version
```

If Sphinx responds then the program is already installed on your machine. Otherwise visit the Sphinx website.

48.3 Removing old builds of the docs

After installing Sphinx, change to the Abjad `docs` directory and use the Sphinx makefile to remove any existing `docs/_build` directory prior to making a new build of the docs.

```
abjad$ cd docs
```

```
docs$ make clean
rm -rf _build/*
```

48.4 Generating the Abjad API

The `docs/scr` directory includes a script to generate the Abjad API. Run this script before building the Abjad docs for the first time.

```
docs$ scr/make-abjad-api
Building TOC tree ...
Now making Sphinx TOC ...

... Done.

Now building the HTML docs ...

sphinx-build -b html -d _build/doctrees   . _build/html
Running Sphinx v1.0.7
loading pickled environment... done

... (many lines omitted) ...

Build finished. The HTML pages are in _build/html.
```

Rerun `make-abjad-api` any time you add or remove a public class, method or function from the codebase.

48.5 Building the HTML docs

Change to the Abjad `docs` directory and run `make html`.

```
abjad$ cd docs
```

```
docs$ make html
sphinx-build -b html -d _build/doctrees   . _build/html
Running Sphinx v1.0.7
loading pickled environment... not found
building [html]: targets for 568 source files that are out of date
updating environment: 568 added, 0 changed, 0 removed
reading sources... [ 13%] chapters/api/debug/debugghandlertoregatorsg
reading sources... [ 37%] chapters/api/tools/clonewp/by_leaf_counts_with_parenta
reading sources... [ 38%] chapters/api/tools/clonewp/by_leaf_range_with_parentag
reading sources... [ 38%] chapters/api/tools/componenttools/get_duration_crosser
reading sources... [ 38%] chapters/api/tools/componenttools/get_duration_preprol
reading sources... [ 39%] chapters/api/tools/componenttools/get_le_duration_prol

... (many more lines omitted) ...

writing output... [ 85%] chapters/api/tools/spannertools/give_attached_to_childr
writing output... [ 95%] chapters/fundamentals/duration/interfaces_compared/inde
writing output... [100%] index                          /indexdexexexng/indexxdexindex
writing additional files... genindex modindex search
copying images... done
copying static files... done
dumping search index... done
dumping object inventory... done
build succeeded.
```

```
Build finished. The HTML pages are in _build/html.
```

You will then find the complete HTML version of the docs in `docs/_build/html`.

```
docs$ ls _build/
doctrees html
```

The output from Sphinx is verbose the first time you build the docs. On sequent builds, Sphinx reports changes only.

```
docs$ make html
sphinx-build -b html -d _build/doctrees    . _build/html
Running Sphinx v1.0.7
loading pickled environment... done
building [html]: targets for 1 source files that are out of date
updating environment: 0 added, 1 changed, 0 removed
reading sources... [100%] chapters/devel/documentation/index
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] index                          ation/index
writing additional files... genindex modindex search
copying static files... done
dumping search index... done
dumping object inventory... done
build succeeded.

Build finished. The HTML pages are in _build/html.
```

48.6 Building a PDF of the docs

Building a PDF of the docs is a two-step process. First you build a LaTeX version of the docs. Then you typeset the LaTeX docs as a PDF.

First change to the Abjad docs directory.

```
abjad$ docs
```

Then make LaTeX sources of the docs.

```
docs$ make latex
sphinx-build -b latex -d _build/doctrees   . _build/latex
Running Sphinx v1.0.7
loading pickled environment... done
building [latex]: all documents
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
processing Abjad.tex... index chapters/start_here/abjad/index chapters/examples/bartok...

(... many lines omitted ...)

...ndices/pitch_conventions/images/example-3.png chapters/examples/ligeti/images/desordre.jpg
copying TeX support files... done
build succeeded.

Build finished; the LaTeX files are in _build/latex.
Run `make all-pdf' or `make all-ps' in that directory to run these through (pdf)latex.
```

Now follow the instructions provided by Sphinx and change to the LaTeX build directory.

```
docs$ cd _build/latex/
```

Then make a PDF version of the docs from the LaTeX sources.

```
latex$ make all-pdf

pdflatex  'Abjad.tex'
This is pdfTeXk, Version 3.141592-1.40.3 (Web2C 7.5.6)
 %&-line parsing enabled.
entering extended mode
(./Abjad.tex
LaTeX2e <2005/12/01>
Babel <v3.8h> and hyphenation patterns for english, usenglishmax, dumylang, noh
yphenation, arabic, basque, bulgarian, coptic, welsh, czech, slovak, german, ng
erman, danish, esperanto, spanish, catalan, galician, estonian, farsi, finnish,

(... many lines omitted ...)
```

The resulting docs will appear as `Abjad.pdf` in the LaTeX build directory you're currently in.

48.7 Building a coverage report

Change to the Abjad `docs` directory and call `sphinx-build` explicitly with the coverage builder, source directory and target directory.

```
docs$ sphinx-build -b coverage . _build/coverage
Making output directory...
Running Sphinx v1.0.7
loading pickled environment... not found
building [coverage]: coverage overview
updating environment: 568 added, 0 changed, 0 removed
reading sources... [ 37%] chapters/api/tools/clonewp/by_leaf_counts_with_parenta
reading sources... [ 38%] chapters/api/tools/clonewp/by_leaf_range_with_parentag
reading sources... [ 38%] chapters/api/tools/componenttools/get_duration_crosser

... (many lines omitted) ...

reading sources... [ 85%] chapters/api/tools/spannertools/withdraw_from_containe
reading sources... [ 95%] chapters/fundamentals/duration/interfaces_compared/ind
reading sources... [100%] index                       t/indexdexexexng/indexxdexindex
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
build succeeded.
```

The coverage report is now available in the `docs/_build/coverage` directory.

```
docs$ ls _build/
coverage doctrees html
```

48.8 Building other versions of the docs

Examine the Sphinx makefile in the Abjad `docs/` directory or change to the `docs/` directory and type `make` with no arguments to see a list of the other versions of the Abjad docs that are available to build.

```
docs$ make
Please use `make <target>' where <target> is one of
    html        to make standalone HTML files
    dirhtml   to make HTML files named index.html in directories
```

```
    pickle    to make pickle files
    json        to make JSON files
    htmlhelp  to make HTML files and a HTML help project
    qthelp    to make HTML files and a qthelp project
    latex     to make LaTeX files, you can set PAPER=a4 or PAPER=letter
    changes   to make an overview of all changed/added/deprecated items
    linkcheck to check all external links for integrity
    doctest   to run all doctests embedded in the documentation (if enabled)
```

48.9 Inserting images with `abjad-book`

Use *abjad-book* to insert snippets of notation in the docs you write in reST.

Embed Abjad code between open and close <abjad> </abjad> tags in your `.rst.raw` sourcefile and then call `abjad-book` to create a pure `.rst` file.

```
abjad-book foo.rst.raw foo.rst

Parsing file ...
Rendering "example-1.ly" ...
Rendering "example-2.ly" ...
```

You will need to build the HTML docs again to see your work.

```
make html
```

48.10 Updating Sphinx

It is important periodically to update your version of Sphinx. If you used `easy_install` to install Sphinx then the usual command to update Sphinx is this:

```
$ sudo easy_install -U Sphinx
```

This will usually work. But if Sphinx fails to update then it may be because you have multiple versions of Python installed on your computer. (This tends especially to be the case under Apple's OS X.)

To get around this first note the version of Python you're currently running:

```
$ python --version
Python 2.6.1
```

Then use a version-explicit form of `easy_install` to update Sphinx:

```
$ sudo easy_install-2.6 -U Sphinx
```

CHAPTER

FORTYNINE

TESTS

Abjad includes an extensive battery of tests. Abjad is in a state of rapid development and extension. Major refactoring efforts are common every six to eight months and are likely to remain so for several years. And yet Abjad continues to allow the creation of complex pieces of fully notated score in the midst of these changes. We believe this is due to the extensive coverage provided by the automated regression battery described in the following sections.

49.1 Automated regression?

A battery is any collection of tests. Regression tests differ from other types of test in that they are designed to be run again and again during many different stages of the development process. Regression tests help ensure that the system continues to function correctly as developers make changes to it. An automated regression battery is one that can be run automatically by some sort of driver with minimal manual intervention.

Several different test drivers are now in use in the Python community. Abjad uses py.test. The `py.test` distribution is not included in the Python standard library, so one of the first thing new contributors to Abjad should do is download and install `py.test`, and then run the existing battery.

49.2 Running the battery

Change to the directory where you have Abjad installed. Then run `py.test`.

```
abjad$ py.test
============================= test session starts ==================================
platform darwin -- Python 2.6.1 -- pytest-2.1.0
collected 4235 items

core/LilyPondContextProxy/test/test_LilypondContextProxy___eq__.py .
core/LilyPondContextProxy/test/test_LilypondContextProxy___repr__.py .
core/LilyPondContextProxy/test/test_LilypondContextProxy___setattr__.py ..

... (many lines omitted) ...

tools/voicetools/test/test_voicetools_iterate_semantic_voices_in_expr.py .
tools/voicetools/test/test_voicetools_iterate_voices_backward_in_expr.py .
tools/voicetools/test/test_voicetools_iterate_voices_in_expr.py .

========================== 4235 passed in 127.06 seconds ===========================
```

Abjad r4629 includes 4235 tests.

49.3 Reading test output

`py.test` crawls the entire directory structure from which you call it, running tests in alphabetical order. `py.test` prints the total number of tests per file in square brackets and prints test results as a single `.` dot for success or else an `F` for failure.

49.4 Writing tests

Project check-in standards ask that tests accompany all code committed to the Abjad repository. If you add a new function, class or method to Abjad, you should add a new test file for that function, class or method. If you fix or extend an existing function, class or method, you should find the existing test file that covers that code and then either add a completely new test to the test file or else update an existing test already present in the test file.

49.5 Test files start with `test_`

When `py.test` first starts up it crawls the entire directory structure from which you call it prior to running a single test. As `py.test` executes this preflight work, it looks for any files beginning or ending with the string `test` and then collects and alphabetizes these. Only after making such a catalog of tests does `py.test` begin execution. This collect-and-cache behavior leads to the important point about naming, below.

49.6 Avoiding name conflicts

Note that the names of **test functions** must be absolutely unique across the entire directory structure on which you call `py.test`. You must never share names between test functions. For example, you must not have two tests named `test_grob_handling_01()` **even if both tests live in different test files**. That is, a test named `test_grob_handling_01()` living in the file `test_accidental_grob_handling.py` and a second test named `test_grob_handling_01()` living in the file `test_notehead_grob_handling.py` will conflict with the each other when `py.test` runs. And, unfortunately, **"py.test is silent about such conflicts when it runs**. That is, should you run `py.test` with the duplicate naming situation described here, what will happen is that `py.test` will correctly run and report results for the **first** such test it finds. However, when `py.test` encounters the second like-named test, `py.test` will incorrectly report cached results for the **first** test rather than the second. The take-away is to include some sort of namespacing indicators in every test name and not to be afraid of long test names. The `test_grob_handling_01()` example given here fixes easily when the two tests rename to `test_accidental_grob_handling_01()` and `test_notehead_grob_handling_01()`.

49.7 Updating `py.test`

It is important periodically to update `py.test`.

The usual command to do this is:

```
$ sudo easy_install -U pytest
```

Note that `pytest` is here spelled without the intervening period.

49.8 Running `doctest` on the `tools` directory

The Python standard library includes the `doctest` module as way of checking the correctness of examples included in Python docstrings. The module searches for instances of the Python interpreter prompt `'>>>'` and executes any code that follows. Abjad docs display the Abjad prompt `'abjad>'` instead of the Python prompt.

This means that all instances of the Abjad prompt must be changed to Python prompts before running `doctest` on the Abjad codebase. Three scripts in `abjad/scr/devel` help do this.

First change to the subdirectory of the Abjad source tree on which you'd like to run `doctest`. Then run these scripts:

```
replace-abjad-prompts-with-python-prompts
```

```
run-doctest-on-all-modules-in-tree
```

```
replace-python-prompts-with-abjad-prompts
```

After running `run-doctest-on-all-modules-in-tree` you can inspect the results that come back from `doctest` and make any fixes as required.

CHAPTER

FIFTY

SCRIPTS

The `abjad/scr/devel` directory contains scripts for Abjad developers. Add `abjad/scr/devel` to your `PATH` to use the scripts described below.

```
abjad$ ls scr/devel
abj-grep                                        find-multifunction-modules
abj-grp                                         find-multiline-import-statements
abj-rmpycs                                      find-nonalphabetized-module-headers
abj-src-grp                                     find-nontrivial-subdirectories
abj-test-grp                                    find-public-helpers-without-docstrings
abj-update                                      find-undocumented-tools
capitalize-test-file-names                      fix-nonalphabetized-module-headers
conjoin-multiline-import-statements             fix-test-case-block-comments
count-source-lines                              fix-test-case-names
count-tools                                     fix-test-case-numbers
duplicate-test-file                             format-lilypond-context-names-with-underscores
find-and-fix-manual-class-package-initializers list-private-modules
find-duplicate-module-names                     rebuild-docs
find-duplicate-tool-module-names                reindent-3-spaces-as-4
find-import-as-statements                       reindent-4-spaces-as-3
find-local-import-statements                    reindent-spaces-variably
find-lower-camel-case-definitions               remove-tmp-out-directories
find-lower-camel-case-modules                   rename-public-helper
find-manual-class-loads-in-initializers         replace-abjad-prompts-with-python-prompts
find-misnamed-private-modules                   replace-in-files
find-missing-test-modules                       replace-python-prompts-with-abjad-prompts
find-module-headers                             run-doctest-on-all-modules-in-tree
find-modules-with-chevrons
```

50.1 Searching the Abjad codebase with `abj-grep`

Abjad provides a wrapper around UNIX `grep` in the form of `abj-grep`. Use this script to recursively search the entire Abjad codebase, leaving out non-human-readable files, files located in special `.svn` Subversion subdirectories, and all files in the `abjad/documentation` directories. You can run `abj-grep` from any directory on your system; you needn't be in the Abjad source directories when you call `abj-grep`.

```
$ abj-grep 'is_assignable('
leaf/duration.py:111:            if not durationtools.is_assignable(rational):
tempo/indication.py:67:            assert durationtools.is_assignable(arg)
tools/check/are_scalable.py:12:            if not durationtools.is_assignable(candidate_duration)
tools/durationtools/is_assignable.py:5:def is_assignable(duration):
tools/durationtools/prolated_to_written.py:2:from abjad.tools.durationtools.is_assignable import
tools/durationtools/prolated_to_written.py:15:    if is_assignable(prolated_duration):
tools/tietools/duration_change.py:28:    if durationtools.is_assignable(new_written_duration):
tools/tuplettools/contents_scale.py:30:    if durationtools.is_assignable(multiplier):
```

50.2 Removing old `*.pyc` files with `abj-rmpycs`

See the section on `abj-update` below for the reasons that it is a good idea to periodically remove the byte-compiled `*.pyc` files that Python generates for its own use behind the scenes. Abjad supplies `abj-rmpycs` to delete all the `*.pyc` in the Abjad codebase, leaving other `*.pyc` on your system untouched.

50.3 Updating your development copy of Abjad with `abj-update`

The normal way of updating your working copy of a Subversion repository is with the `svn update` or `svn up` command. You can update your working copy of Abjad in the usual way with `svn up`. But Abjad supplies an `abj-update` script as a wrapper around the usual Subversion update commands. In addition to updating your working copy of Abjad, `abj-update` populates the `abjad/.version` file with the most recent revision number of the system, and then removes all `*.pyc` files from your Abjad install. The benefits here are twofold. First, Abjad adds the most recent revision number of the system to all `.ly` files that you generate when working with Abjad. If you do not update the Abjad version file on a regular basis, the headers in your Abjad-generated `.ly` files will list the wrong version of the system. Second, as is the case in working with any substantial Python codebase, it is a good idea to periodically remove the byte-compiled `*.pyc` files that Python creates for its own use. The reason for this is inadvertant name aliasing. That is, if there was previously a module named `foo.py` somewhere in the system and if Python had at some point imported the module and created `foo.pyc` as a byprodct, this `.pyc` file will remain on the filesystem even if you later decide to remove, or rename, the source `foo.py` module. This lead to confusion because days or weeks after `foo.py` has been removed, Python will still find `foo.pyc` and seem to make the contents of `foo.py` available from beyond the grave. Updating with `abj-update` takes care of these two situations.

50.4 Counting lines of code with `count-source-lines`

Run `count-source-lines` for a count of lines of count divided between source and test files.

```
abjad$ count-source-lines

source_modules: 1703
test_modules:   1812

source_lines:   73942
test_lines:     76636

total lines:    150578
test-to-source ratio is 1 : 1
```

The script is directory-dependent so you can run it any the entire Abjad codebase or any subdirectory of the codebase.

50.5 Global search-and-replace with `replace-in-files`

You probably won't need to use `replace-in-files` very often. But if you are making changes to Abjad that will cause some name, such as `FooBar`, to be globally changed everywhere in the Abjad codebase to, say to `foo_bar`, then you can use `replace-in-files` to save lots of time.

```
$ replace-in-files --help

    Usage:

    replace-in-files DIR OLD_TEXT NEW_TEXT [CONFIRM=true/false]

    Crawl directory DIR and read every file in it recursively.
```

```
    Replace OLD_TEXT with NEW_TEXT in each file.

    Set CONFIRM to `false` to replace without prompting.
```

50.6 Adding new development scripts

If you write and then find yourself using a certain script over and over again when you're developing new code for Abjad, consider contributing back to the project so we can include your script in the next public release of Abjad. Scripts in the the Abjad script directories end with no file extension and try to be as OS-portable as possible, which usually means writing the script in Python, rather than your operating system's shell, and relying heavily on Python's `os` module.

CHAPTER

FIFTYONE

USING ABJAD-BOOK

abjad-book is an independent application included in every installation of Abjad. abjad-book allows you to write Abjad code in the middle of documents written in HTML, LaTeX or ReST. We created abjad-book to help us document Abjad. Our work on abjad-book was inspired by lilypond-book, which does for LilyPond much what abjad-book does for Abjad.

51.1 HTML with embedded Abjad

To see abjad-book in action, open a file and write some HTML by hand. Add some Abjad code to your HTML between open and close <abjad> </abjad> tags.

```
<html>

<p>This is an <b>HTML</b> document.</p>

<p>The code is standard hypertext mark-up.</p>

<p>Here is some music notation generated automatically by Abjad:</p>

<abjad>
v = Voice("c'8 d' e' f' g' a' b' c''")
beam = beamtools.BeamSpanner(v)
show(v)
</abjad>

<p>And here is more ordinary <b>HTML</b>.</p>

</html>
```

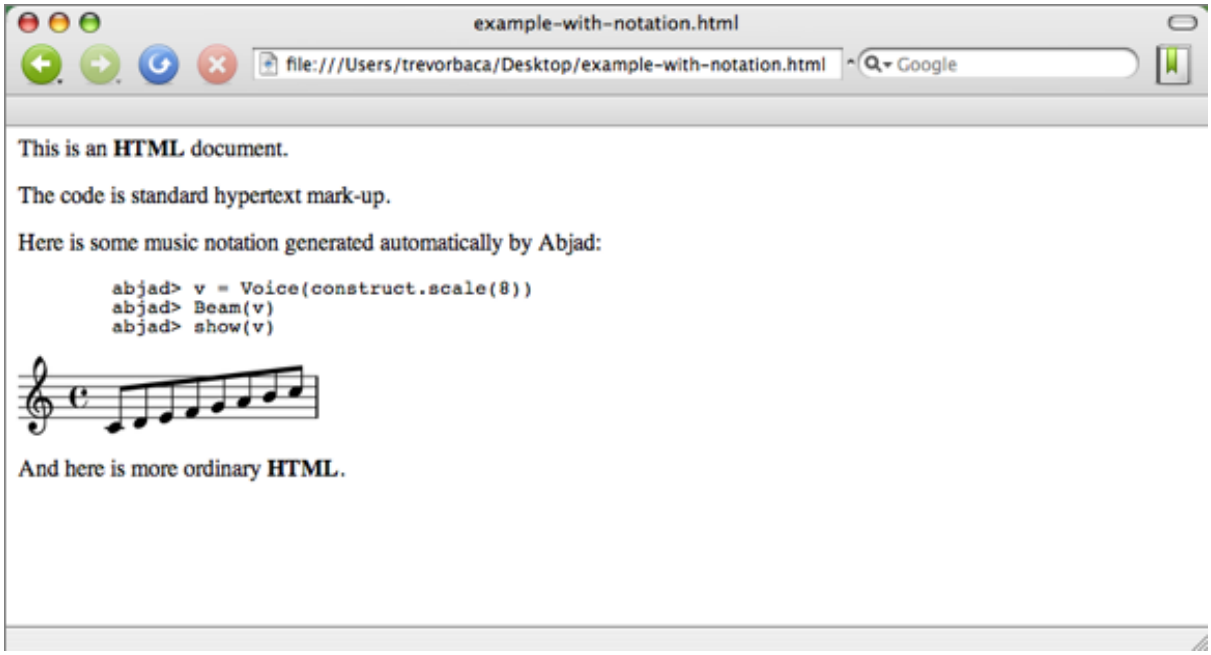
Save your the file with the name example.html.raw. You now have an HTML file with embedded Abjad code.

In the terminal, call abjad-book on example.html.raw.

```
$ abjad-book example.html.raw example.html

Parsing file...
Rendering "abjad-book-1.ly"...
```

The application opens example.html.raw, finds all Abjad code between <abjad> </abjad> tags, executes it, and then creates and inserts image files of music notation accordingly.

Open example.html with your browser.



That's all there is to it. `abjad-book` lets you open a file and type HTML by hand with Abjad sandwiched between the special <abjad> </abjad> tags described here. Run `abjad-book` on such a hybrid file to create pure HTML with images of music notation created by Abjad.

Note: `abjad-book` makes use of ImageMagick's convert application to crop and scale PNG images generated for HTML and ReST documents. For LaTeX documents, `abjad-book` uses `pdfcrop` for cropping PDFs.

51.2 LaTeX with embedded Abjad

You can use `abjad-book` to insert Abjad code and score excerpts into any LaTeX you create. Type the sample code below into a file.

```
\documentclass{article}
\usepackage{graphicx}
\usepackage{listings}
\begin{document}

This is a standard LaTeX document with embedded Abjad.

The code below creates an Abjad measure and then prints the measure
format string.

<abjad>
measure = Measure((5, 8), "c'8 d'8 e'8 f'8 g'8")
f(measure)
</abjad>

This next bit of code knows about the measure we defined earlier.

<abjad>
iotools.write_expr_to_ly(measure, 'abjad-book-1', docs=True) <hide
</abjad>

And this is the end of the our sample LaTeX document.

\end{document}
```

Save your file with the name `example.tex.raw`. You now have a LaTeX file with embedded Abjad code.

In the terminal, call `abjad-book` on `example.tex.raw`.

```
$ abjad-book example.tex.raw example.tex

Processing 'example.tex.raw'. Will write output to 'example.tex'...
Parsing file...
Rendering "abjad-book-1.ly"...
```

The application open `example.tex.raw`, finds all code between Abjad tags, executes it, and then creates and inserts Abjad interpreter output and PDF files of music notation. You can view the contents of the next LaTeX file `abjad-book` has created.

```
\documentclass{article}
\usepackage{graphicx}
\usepackage{listings}
\begin{document}

This is a standard LaTeX document with embedded Abjad.

The code below creates an Abjad measure and then prints the measure
format string.

\begin{lstlisting}[basicstyle=\footnotesize, tabsize=4, showtabs=false, showspaces=false]
   >>> measure = Measure((5, 8), "c'8 d'8 e'8 f'8 g'8")
   >>> f(measure)
   {
      \time 5/8
      c'8
      d'8
      e'8
      f'8
      g'8
   }
\end{lstlisting}

This next bit of code knows about the measure we defined earlier.
This code renders the measure as a PDF using a template suitable
for inclusion in LaTeX documents.

\includegraphics{images/abjad-book-1.pdf}

And this is the end of the our sample LaTeX document.

\end{document}
```

You can now process the file `example.tex` just like any other LaTeX file, using `pdflatex` or TexShop or whatever LaTeX compilation program you normally use on your computer.

```
$ pdflatex example.tex

This is pdfTeXk, Version 3.141592-1.40.3 (Web2C 7.5.6)
 %&-line parsing enabled.
entering extended mode
...
```

And then open the resulting PDF.

51.3 Using abjad-book on ReST documents

You can call abjad-book on ReST documents, too. Follow the examples given here for HTML and LaTeX documents and modify accordingly.

51.4 Using [hide = True]

You can add [hide = True] to any abjad-book example to show only music notation.

```
<abjad>[hide = True]
staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b''8")
iotools.write_expr_to_ly(staff, 'staff-example', docs=True)
</abjad>
```

CHAPTER

FIFTYTWO

TIMING CODE

You can time code with Python's built-in `timeit` module:

```
from abjad import *
import timeit

timer = timeit.Timer('Note(0, (1, 4))', 'from __main__ import Note')
print timer.timeit(1000)
```

```
0.225436925888
```

These results show that 1000 notes take 0.23 seconds to create.

Other Python timing modules are available for download on the public Internet.

CHAPTER

FIFTYTHREE

PROFILING CODE

Profile code with `profile_expr()` in the `iotools` package:

```
>>> iotools.profile_expr('Note(0, (1, 4))')
Sun Aug 14 16:50:36 2011    _tmp_abj_profile

         327 function calls (312 primitive calls) in 0.001 CPU seconds

   Ordered by: cumulative time
   List reduced from 96 to 12 due to restriction <12>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.001    0.001 <string>:1(<module>)
        1    0.000    0.000    0.001    0.001 Note.py:18(__init__)
        1    0.000    0.000    0.001    0.001 Note.py:133(fset)
        1    0.000    0.000    0.001    0.001 NoteHead.py:18(__init__)
        1    0.000    0.000    0.001    0.001 NoteHead.py:121(fset)
        1    0.000    0.000    0.001    0.001 NamedChromaticPitch.py:28(__new__)
        1    0.000    0.000    0.000    0.000 Leaf.py:18(__init__)
        1    0.000    0.000    0.000    0.000 chromatic_pitch_name_to_diatonic_pitch_numbe
        1    0.000    0.000    0.000    0.000 octave_tick_string_to_octave_number.py:4(oct
        1    0.000    0.000    0.000    0.000 re.py:134(match)
        1    0.000    0.000    0.000    0.000 re.py:227(_compile)
        1    0.000    0.000    0.000    0.000 sre_compile.py:501(compile)
```

These results show 327 function calls to create a note.

The `profile_expr()` function wraps the Python `cProfile` and `pstats` modules.

CHAPTER

FIFTYFOUR

MEMORY CONSUMPTION

You can examine memory consumption with tools included in the `guppy` module:

```
from guppy import hpy
hp = hpy()
hp.setrelheap()
notes = [Note(0, (1, 4)) for x in range(1000)]
h = hp.heap()
print h
```

```
Partition of a set of 11024 objects. Total size = 586364 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0   1000   9   124000  21    124000  21 abjad.tools.notetools.Note.Note.Note
     1   1004   9   116464  20    240464  41 __builtin__.set
     2   2003  18    76300  13    316764  54 list
     3   1000   9    52000   9    368764  63
                                             abjad.tools.pitchtools.NamedChromaticPitch.NamedChro
                                             icPitch.NamedChromaticPitch
     4   1000   9    44000   8    412764  70
                                             abjad.interfaces._OffsetInterface._OffsetInterface._
                                             setInterface
     5   1000   9    44000   8    456764  78 abjad.tools.notetools.NoteHead.NoteHead.NoteHead
     6   1000   9    40000   7    496764  85 0x23add0
     7   1000   9    32000   5    528764  90
                                             abjad.interfaces.ParentageInterface.ParentageInterfa
                                             ParentageInterface
     8   1011   9    28568   5    557332  95 str
     9   1000   9    28000   5    585332 100
                                             abjad.interfaces._NavigationInterface._NavigationInt
                                             ace._NavigationInterface
<6 more rows. Type e.g. '_.more' to view.>
```

These results show 586K for 1000 notes.

You must download `guppy` from the public Internet because the module is not included in the Python standard library.

CHAPTER

FIFTYFIVE

CLASS ATTRIBUTES

Consider the definition of this class:

```
class FooWithInstanceAttribute(object):

   def __init__(self):
      self.constants = (
         'red', 'orange', 'yellow', 'green',
         'blue', 'indigo', 'violet',
         )
```

1000 objects consume 176k:

```
from guppy import hpy
hp = hpy()
hp.setrelheap()
objects = [FooWithInstanceAttribute() for x in range(1000)]
h = hp.heap()
print h
```

```
Partition of a set of 2004 objects. Total size = 176536 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0   1000  50   140000  79    140000  79 dict of __main__.FooWithInstanceAttribute
     1   1000  50    32000  18    172000  97 __main__.FooWithInstanceAttribute
     2      1   0     4132   2    176132 100 list
     3      1   0      348   0    176480 100 types.FrameType
     4      1   0       44   0    176524 100 __builtin__.weakref
     5      1   0       12   0    176536 100 int
```

But consider the definition of this class:

```
class FooWithSharedClassAttribute(object):

   def __init__(self):
      pass

   self.constants = (
      'red', 'orange', 'yellow', 'green',
      'blue', 'indigo', 'violet',
      )
```

1000 objects consume only 36k:

```
from guppy import hpy
hp = hpy()
hp.setrelheap()
objects = [FooWithClassAttribute() for x in range(1000)]
h = hp.heap()
print h
```

```
Partition of a set of 1004 objects. Total size = 36536 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0   1000 100    32000  88     32000  88 __main__.FooWithClassAttribute
     1      1   0     4132  11     36132  99 list
```

```
     2      1   0      348   1     36480 100 types.FrameType
     3      1   0       44   0     36524 100 __builtin__.weakref
     4      1   0       12   0     36536 100 int
```

Objects that share class attributes between them can consume less memory than objects that don't. But consider the usual provisions between class attributes and instance attributes when implementing custom classes. Class attributes make sense when objects will never modify the attribute in question. Class attributes also make sense when objects will modify the attribute in question and will desire to change the attribute in question for all other like objects at the same time. Probably best to use instance attributes in most other cases.

CHAPTER

FIFTYSIX

USING SLOTS

Consider the definition of this class:

```
class Foo(object)

   def __init__(self, a, b, c):
      self.a = a
      self.b = b
      self.c = c
```

1000 objects consume 176k:

```
from guppy import hpy
hp = hpy()
hp.setrelheap()
objects = [Foo(1, 2, 3) for x in range(1000)]
h = hp.heap()
print h
```

```
Partition of a set of 2004 objects. Total size = 176536 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0   1000  50   140000  79    140000  79 dict of __main__.FooWithInstanceAttribute
     1   1000  50    32000  18    172000  97 __main__.FooWithInstanceAttribute
     2      1   0     4132   2    176132 100 list
     3      1   0      348   0    176480 100 types.FrameType
     4      1   0       44   0    176524 100 __builtin__.weakref
     5      1   0       12   0    176536 100 int
```

But consider the definition of this class:

```
class FooWithSlots(object):

   __slots__ = ('a', 'b', 'c')
   def __init__(self, a, b, c):
      self.a = a
      self.b = b
      self.c = c
```

1000 objects consume only 40k:

```
from guppy import hpy
hp = hpy()
hp.setrelheap()
objects = [FooWithSlots(1, 2, 3) for x in range(1000)]
h = hp.heap()
print h
```

```
Partition of a set of 1004 objects. Total size = 40536 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0   1000 100    36000  89     36000  89 __main__.Bar
     1      1   0     4132  10     40132  99 list
     2      1   0      348   1     40480 100 types.FrameType
```

```
     3      1   0       44   0     40524 100 __builtin__.weakref
     4      1   0       12   0     40536 100 int
```

The example here confirms the Python Reference Manual 3.4.2.4: “By default, instances of both old and new-style classes have a dictionary for attribute storage. This wastes space for objects having very few instance variables. The space consumption can become acute when creating large numbers of instances.”

CHAPTER

FIFTYSEVEN

CODING STANDARDS

Indent with spaces, not with tabs. Use four spaces at a time:

```
def foo(x, y):
    return x + y
```

Introduce comments with one pound sign and a single space:

```
# comment before foo
def foo(x, y):
    return x + y
```

Avoid `from`. Instead of `from fractions import Fraction` use:

```
import fractions
```

Favor early imports at the head of each module. Only one `import` per line.

Arrange standard library imports alphabetically at the head of each module:

```
import fractions
import types
```

Follow standard library imports with intrapackage Abjad imports arranged alphabetically:

```
import footools
import bartools
import blahtools
```

Include two blank lines after `import` statements before the rest of the module:

```
import fractions
import types
import footools
import bartools
import blahtools


class Foo(object):
    ...
    ...
```

Wrap docstrings with triple apostrophes and align like this:

```
def foo(x, y):
    '''This is the first line of the foo docstring.
    This is the second line of the foo docstring.
    And this is the last line of the foo docstring.
    '''
```

Use paired apostrophes to delimit strings:

```
s = 'foo'
```

Use paired quotation marks to delimit strings within a string:

```
s = 'foo and "bar"'
```

Name classes in upper camelcase:

```
def FooBar(object):
    ...
    ...
```

Name bound methods in underscore-delimited lowercase:

```
def Foo(object):

    def bar_blah(self):
        ...

    def bar_baz(self):
        ...
```

Name module-level functions in underscore-delimited lowercase:

```
def foo_bar():
    ...

def foo_blah():
    ...
```

Separate bound method definitions with a single empty line:

```
class FooBar(object):

    def __init__(self, x, y):
        ...

    def bar_blah(self):
        ...

    def bar_baz(self):
        ...
```

Organize the definitions of core classes into the nine following major sections:

```
class FooBar(object):

    ### CLASS ATTRIBUTES ###

    special_enumeration = ('foo', 'bar', 'blah')

    ### INITIALIZER ###

    def __init__(self, x, y):
        ...

    ### SPECIAL METHODS ###

    def __repr__(self):
        ...

    def __str__(self):
        ...

    ### READ-ONLY PRIVATE PROPERTIES ###

    @property
    def _foo(self):
        ...

    ### READ / WRITE PRIVATE PROPERTIES ###

    @apply
    def _bar():
        def fget(self):
            ...
```

```
        def fset(self, expr):
            ...
        return property(**locals())

    ### PRIVATE METHODS ###

    def _blah(self, x, y):
        ...

    ### READ-ONLY PUBLIC PROPERTIES ###

    @property
    def foo(self):
        ...

    ### READ / WRITE PUBLIC PROPERTIES ###

    @apply
    def bar():
        def fget(self):
            ...
        def fset(self, expr):
            ...
        return property(**locals())

    ### PUBLIC METHODS ###

    def blah(self, expr):
        ...
```

Use < less-than signs in preference to greater-than signs:

```
if x < y < z:
    ...
```

Preceed private class attributes with a single underscore.

Alphabetize method names.

Alphabetize keyword arguments.

Include keyword argument names explicitly in function calls.

Limit docstring lines to 99 characters.

Limit source lines to 110 characters and use `\` to break lines where necessary.

Eliminate trivial slice indices. Use `s[:4]` instead of `s[0:4]`.

Prefer new-style string formatting to old-style string interpolation. Use `'string {} content'.format(expr)` instead of `'string %s content' % expr`.

Prefer list comprehensions to `filter()`, `map()` and `apply()`.

Do not abbreviate variable names. (But use `expr` for 'expression' and use `i` or `j` for loop counters.)

Name variables that represent a list or other collection of objects in the plural.

Name functions beginning with a verb. (But use `noun_to_noun` for conversion functions and `mathtools.noun` for some `mathtools` functions.)

Avoid private classes.

Avoid private functions. (But use private class methods as necessary.)

Implement package-level functions in preference to `staticmethod` class methods.

Implement only one statement per line of code.

Implement only one class per module.

Implement only one function per module.

Author one `py.test` test file for every module-level function.

Author one `py.test` test file for every bound method in the public interface of a class.

Part VII

Appendices

CHAPTER

FIFTYEIGHT

FROM TREVOR AND VÍCTOR

We are composers Trevor Bača and Víctor Adán, creators of Abjad, and our earliest collaborative work dates back to shared undergraduate years in Austin. It was the mid- to late-90s and we found ourselves interested in ways of building up ever larger sets of musical materials in our scores, with ever greater amounts of musical information.

Our work then began with pitch formalization, creating materials in C and then writing the results as MIDI to hear what we'd created. Turns out that this is a fairly common gateway into materials generation for many composers, and so it was for us. Probably this was, and is, due to the ever present availability of MIDI and, to a lesser extent, CSound. But even back then it was clear to us to finding ways to embody other aspects of the musical score – from nested rhythms to the different approaches to the musical measure to the arbitrarily complex structures possible with overlapping musical voices – would require a wholly different level of consideration, and different development techniques as well.

As an example, consider flat lists of floating-point values. This basic data structure, together with the constant need some type of quantification or rounding, feeds much of most composers' work with CSound, pd and the like. It is a good thing, therefore, that essentially all modern programming languages include tools for manipulating flat lists of floats out of the box, or in the standard library. But what happens when you want to think of pitch as something much more than integers for core values with, perhaps, floats for microtones? What if you want to work with pitches as fully-fledged objects? Objects capable of carrying arbitrarily large sets of attributes and values? Objects that might group together, first into sets, and then into larger assemblages, and then into still larger complexes of pitch information loaded, or even overloaded, with cross-relationships or textural implications? Carrying this surplus of information about pitch, or the potential uses of pitch, in data structures limited to, or centered around, the list-of-floats paradigm then becomes a burden.

And what of working with rhythms not only as offset values, as implied by the list-of-floats approach, but as arbitrarily nested, stretched, compressed and stacked sets of values, as allowed by the tupleting and measure structures of conventional score? A different approach is needed.

There was, and still is, no reason to believe that general purpose programming languages and development tools should come readily supplied with the objects and methods most suitable for composerly applications. And this means that the attributes of a domain-specific language that will best meet the needs of composes interested in working formally with the full complement of capabilities in traditional score remains an open question.

We continued our work in score formalization independenly until 2005, Trevor in a system that would come to be called Lascaux, and Víctor in a system dubbed Cuepatlahto. We experimented with C, Mathematica and Matlab as the core programming languages driving our systems before settling independently on Python, Víctor out of experinece at MIT, where he was working on his masters at the Media Lab with Berry Vercoe, and Trevor out of the working necessities of a professional developer and engineer.

We passed through indepedent experiences using Finale, Sibelius, Leland Smith's SCORE, and even Adobe Illustrator as the notational rendering engines for Lascaux and Cuepatlahto. Through all of this, both systems were designed to tackle a shared set of problems. These included:

1. The difficulty involved in transcribing larger scale and highly parameterized gestures and textures into traditional Western notation.
2. The general inflexbility of closed, commercial music notation software packages.

3. The relative inability of objects on the printed page in conventional score to point to each other — or, indeed, to other objects or ideas outside the printed page — in ways rich enough to help capture, model and develop long-range, nonlocal relationships throughout our scores.

Afer collaborating on a joint paper describing the two systems, and after discussing collaborative design and implementation at length, both online and in weekends' long review of our respective codebases, we decided to combine our efforts into a single, unified project. That project is now Abjad.

In our work on Abjad we strive to develop a powerful and flexible symbolic system. We picked the phrase 'formalized score control', or FSC, as a nod to Xenakis, who was so far ahead in so many ways, and also to highlight our primary project goal: to bring the full power of modern programming languages, and tools in mathematics, text processing, pattern recognition, and modular, iterative and incremental development to bear on all parts of the compositional process.

CHAPTER

FIFTYNINE

WHY MIDI IS NOT ENOUGH

Given that Abjad models written musical score, it might seem odd for MIDI to be even mentioned in this manual. Yet, until fairly recently, MIDI has played a role (sometimes tangential, other times fundamental) in a variety of software tools related to music notation and engraving.

59.1 A very brief overview of MIDI

MIDI (Musical Instrument Digital Interface) was first introduced in 1981 by Dave Smith, the founder of Sequential Circuits. The original purpose of MIDI was to allow the communication between different electronic musical instruments; more specifically, to allow one device to send **control** data to another device. Typical messages might be "note On" (play a *note*) "note Off" (turn off a *note*). A MIDI "note" message, for example, is composed of three bytes: the first byte (the Status byte) tells the device what kind of message this is (e.g. a Note On message). The second byte encodes key number (which key was pressed) and the third byte, velocity (how hard the key was pressed). It should be clear that a *Note* in this context means something very different than *Note* in the context of a traditional printed score. While the bias towards keyboard interfaces is clear in the definition of the MIDI Note control message, one can still give the MIDI note a more general use by reinterpreting "key number" as pitch and "velocity" as loudness, the usual perceptual correlates of these control changes as well as the most meaningful musical parameters in western music.

With the subsequent proliferation of music production software, the SMF (Standard Midi File) was introduced to allow the recording and storage of the control data from a MIDI stream. The SMF required a time stamp to keep track of when control messages took place. These are called "delta-times" in the SMF specification.

"The MTrk chunk type is where actual song data is stored. It is simply a stream of MIDI events (and non-MIDI events), preceded by delta-time values."

In combination with the MIDI Note message, the addition of duration now allowed one to have a minimal but sufficient **machine** representation–a machine score–of music requiring only these parameters: duration, pitch and loudness. Such is the case of most piano music.

59.2 Limitations of MIDI from the point of view of score modeling

But, alas, there is much more information in a printed score that can not be practically encoded in a SMF. Common musical notions such as meter, clef, key signature, articulation, to name only a few, are ignored. A desire to include some of these concepts in MIDI is evident in the inclusion of some so called *meta-events*. From the SMF specification: " specifies non-MIDI information useful to this format or to sequencers." Examples of *meta-events* are *Time Signature* and *Key Signature*. In addition to the semantic elements just mentioned, there are also the typographical elements (such as line thickness, spacing, color, fonts, etc.) that all printed scores carry. This extra layer of information is completely absent in a SMF. However, from the point of view of encoding a printed score, the main limitation of MIDI is not the lack musical features or the absence of typographical data, but the assumption that musical durations, pitches and loudnesses can be each fully and efficiently encoded with integers or even fractions. In a printed score, this is not the case for any of them. MIDI encodes only *magnitudes*: time

interval magnitudes, pitch interval magnitudes, velocity magnitudes. While these may be sufficient attributes for an automated piano performance, they are not all the attributes of notes in a printed score.

59.3 Written note durations vs. MIDI delta-times

Assume a fixed tempo has been set. Assume that all magnitudes are represented with (and limited to) rational numbers. A time interval magnitude d = 1/4 has an infinity of equivalent representations in terms of magnitude: d = 1/4 = 1/8 * 2 = 1/8 + 1/16 * 2 ... etc. So, for example, while equivalent in magnitude, these are not the same notated durations:

```
>>> m1 = measuretools.AnonymousMeasure([Note("c'4")])
>>> m2 = measuretools.AnonymousMeasure(Note(0, (1, 8)) * 2)
>>> tietools.TieSpanner(m2)
TieSpanner(|1/4(2)|)
>>> m3 = measuretools.AnonymousMeasure([Note(0, (1, 8))] + Note(0, (1, 16)) * 2)
>>> tietools.TieSpanner(m3)
TieSpanner(|1/4(3)|)
>>> r = stafftools.RhythmicStaff([m1, m2, m3])
```

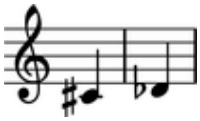
```
>>> show(r)
```

59.4 Written note pitch vs. MIDI note-on

A similar thing happens with pitches. In MIDI, key (pitch) number 61 is a half tone above middle C. But how is this pitch to be notated? As a C sharp or a B flat?

```
>>> m1 = measuretools.AnonymousMeasure([Note(1, (1, 4))])
>>> m2 = measuretools.AnonymousMeasure([Note(('df', 4), (1, 4))])
>>> r = Staff([m1, m2])
```

```
>>> show(r)
```



59.5 Conclusion

MIDI was not designed for score representation. MIDI is a simple communication protocol intended for real-time control. As such, it naturally lacks the adequate model to represent the full range of information found in printed scores.

CHAPTER

SIXTY

WHY LILYPOND IS RIGHT FOR ABJAD

Early versions of Abjad wrote MIDI files for input to Finale and Sibelius. Later versions of Abjad wrote `.pbx` files for input into Leland Smith's SCORE. Over time we found LilyPond superior to Finale, Sibelius and SCORE.

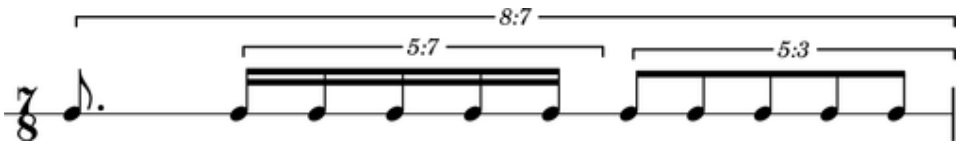
60.1 Nested tuplets works out of the box

LilyPond uses a single construct to nest tuplets arbitrarily:

```
\new stafftools.RhythmicStaff {
    \time 7/8
    \times 7/8 {
        c8.
        \times 7/5 { c16 c16 c16 c16 c16 }
        \times 3/5 { c8 c8 c8 c8 c8 }
    }
}
```

```
>>> staff = stafftools.RhythmicStaff([Measure((7, 8), [])])
>>> measure = staff[0]
>>> measure.append(Note('c8.'))
>>> measure.append(Tuplet(Fraction(7, 5), 5 * Note('c16')))
>>> beamtools.BeamSpanner(measure[-1])
BeamSpanner({c16, c16, c16, c16, c16})
>>> measure.append(Tuplet(Fraction(3, 5), 5 * Note('c8')))
>>> beamtools.BeamSpanner(measure[-1])
BeamSpanner({c8, c8, c8, c8, c8})
>>> Tuplet(Fraction(7, 8), measure.music)
Tuplet(7/8, [c8., {* 5:7 c16, c16, c16, c16, c16 *}, {* 5:3 c8, c8, c8, c8, c8 *}])
>>> staff.override.tuplet_bracket.bracket_visibility = True
>>> staff.override.tuplet_bracket.padding = 1.6
```

```
>>> show(staff, docs=True)
```



LilyPond's tuplet input syntax works the same as any other recursive construct.

60.2 Broken tuplets work out of the box

LilyPond engraves tupletted notes interrupted by nontupletted notes correctly:

```
\new Staff {
    \times 4/7 { c'16 c'16 c'16 c'16 }
    c'8 c'8
```

```
    \times 4/7 { c'16 c'16 c'16 }
}
```

```
>>> t = Tuplet(Fraction(4, 7), Note(0, (1, 16)) * 4)
>>> notes = Note(0, (1, 8)) * 2
>>> u = Tuplet(Fraction(4, 7), Note(0, (1, 16)) * 3)
>>> beamtools.BeamSpanner(t)
BeamSpanner({c'16, c'16, c'16, c'16})
>>> beamtools.BeamSpanner(notes)
BeamSpanner(c'8, c'8)
>>> beamtools.BeamSpanner(u)
BeamSpanner({c'16, c'16, c'16})
>>> measure = Measure((4, 8), [t] + notes + [u])
>>> staff = stafftools.RhythmicStaff([measure])
```

```
>>> show(staff, docs=True)
```

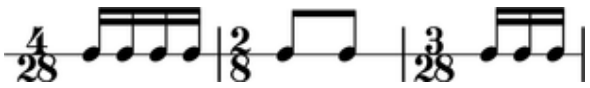
60.3 Nonbinary meters work out of the box

The rhythm above rewrites with time signatures in place of tuplets:

```
\new Staff {
    \time 4/28 c'16 c'16 c'16 c'16 |
    \time 2/8  c'8  c'8 |
    \time 3/28 c'16 c'16 c'16 |
}
```

```
>>> t = Measure((4, 28), Note(0, (1, 16)) * 4)
>>> u = Measure((2, 8), Note(0, (1, 8)) * 2)
>>> v = Measure((3, 28), Note(0, (1, 16)) * 3)
>>> beamtools.BeamSpanner(t)
BeamSpanner(|4/28(4)|)
>>> beamtools.BeamSpanner(u)
BeamSpanner(|2/8(2)|)
>>> beamtools.BeamSpanner(v)
BeamSpanner(|3/28(3)|)
>>> staff = stafftools.RhythmicStaff([t, u, v])
```

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



The time signatures `4/28` and `3/28` here have a denominator not equal to `4`, `8`, `16` or any other nonnegative integer power of two. Abjad calls such time signatures **nonbinary meters** and LilyPond engraves them correctly.

60.4 Lilypond models the musical measure correctly

Most engraving packages make the concept of the measure out to be more important than it should. We see evidence of this wherever an engraving package makes it difficult for either a long note or the notes of a tuplet to cross a barline. These difficulties come from working the idea of measure-as-container deep into object model of the package.

There is a competing way to model the musical measure that we might call the measure-as-background way of thinking about things. Western notation pratice started absent any concept of the barline, introduced the idea gradually, and has since retreated from the necessity of the convention. Engraving packages that pick out an understanding of the barline from the 18th or 19th centuries subscribe to the measure-as-container view of things

and oversimplify the problem. One result of this is to render certain barline-crossing rhythmic figures either an inelegant hack or an outright impossibility. LilyPond eschews the measure-as-container model in favor of the measure-as-background model better able to handle both earlier and later notation practice.

CHAPTER

SIXTYONE

LILYPOND TEXT ALIGNMENT

LilyPond provides many ways to position text.

61.1 Default alignment

LilyPond left-aligns markup relative to the left edge of note heads by default.

```
>>> from abjad.tools import documentationtools
```

```
>>> staff = stafftools.RhythmicStaff('c')
```

```
>>> markuptools.Markup('XX', Up)(staff[0])
Markup(('XX',), direction=Up)(c4)
```

```
>>> lilypond_file = documentationtools.make_text_alignment_example_lilypond_file(staff)
>>> show(lilypond_file)
```

61.2 TextScript #'self-alignment-X

Use #'self-alignment-X to left-, center- or right-align markup relative to the left edge of note heads.

Note that changes to #'self-alignment-X do not change the fact that markup positioning is by default relative to the left edge of note heads.

```
>>> staff = stafftools.RhythmicStaff('c c c')
```

```
>>> markuptools.Markup('XX', Up)(staff[0])
Markup(('XX',), direction=Up)(c4)
>>> staff[0].override.text_script.self_alignment_X = 'left'
>>> markuptools.Markup('XX', Up)(staff[1])
Markup(('XX',), direction=Up)(c4)
>>> staff[1].override.text_script.self_alignment_X = 'center'
>>> markuptools.Markup('XX', Up)(staff[2])
Markup(('XX',), direction=Up)(c4)
>>> staff[2].override.text_script.self_alignment_X = 'right'
```

```
>>> lilypond_file = documentationtools.make_text_alignment_example_lilypond_file(staff)
>>> show(lilypond_file)
```

61.3 TextScript #'X-offset

Use `#'X-offset` to offset markup by some number of magic units in the horizontal direction.

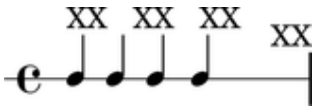
Specify `#'X-offset` arguments as numbers like `#2.5`. Do not specify `#'X-offset` arguments as direction contstants like `#right`.

Note that changes to `#'X-offset` do not change the fact that markup positioning is by default relative to the left edge of note heads.

```
>>> staff = stafftools.RhythmicStaff('c c c c')
```

```
>>> markuptools.Markup('XX', Up)(staff[0])
Markup(('XX',), direction=Up)(c4)
>>> staff[0].override.text_script.X_offset = 0
>>> markuptools.Markup('XX', Up)(staff[1])
Markup(('XX',), direction=Up)(c4)
>>> staff[1].override.text_script.X_offset = 2
>>> markuptools.Markup('XX', Up)(staff[2])
Markup(('XX',), direction=Up)(c4)
>>> staff[2].override.text_script.X_offset = 4
>>> markuptools.Markup('XX', Up)(staff[3])
Markup(('XX',), direction=Up)(c4)
>>> staff[3].override.text_script.X_offset = 6
```

```
>>> lilypond_file = documentationtools.make_text_alignment_example_lilypond_file(staff)
>>> show(staff)
```



CHAPTER

SIXTYTWO

SCORE SNIPPET GALLERY

Abjad uses a collection of score snippets in many places throughout the docs, tests and other parts of the codebase. Some of these are collected here.

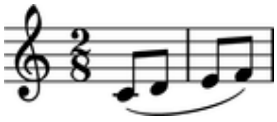
62.1 Score snippet 1

This score features two measures with a beam spanner applied to each measure and a slur spanner applied to all the notes in the score:

```
>>> staff = Staff(r"abj: | 2/8 c'8 d'8 || 2/8 e'8 f'8 |")
>>> beamtools.apply_beam_spanners_to_measures_in_expr(staff)
[BeamSpanner(|2/8(2)|), BeamSpanner(|2/8(2)|)]
>>> spannertools.SlurSpanner(staff.leaves)
SlurSpanner(c'8, d'8, e'8, f'8)
```

```
>>> f(staff)
\new Staff {
    {
        \time 2/8
        c'8 [ (
        d'8 ]
    }
    {
        e'8 [
        f'8 ] )
    }
}
```

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



Score snippet 1 is used widely in the component split tests.

CHAPTER

SIXTYTHREE

CHANGE LOG

63.1 Changes from 2.10 to 2.11

Renamed `timetokentools` package. The new name is `rhythmmakertools`.

Renamed all rhythm maker classes. Replaced `TimeTokenMaker` with `RhythmMaker` everywhere. Replaced `Token` with `Division` everywhere.

Renamed `durationtools.yield_all_assignable_rationals()`. The new name is `durationtools.yield_all_assignable_durations()`.

Renamed `durationtools.rewrite_rational_under_new_tempo()`. The new name is `durationtools.rewrite_duration_under_new_tempo()`.

Renamed `durationtools.rewrite_duration_under_new_tempo()`. The new name is `tempotools.rewrite_duration_under_new_tempo()`.

Renamed `tempotools.integer_tempo_to_tempo_multiplier_pairs()`. The new name is `tempotools.rewrite_integer_tempo()`.

Renamed `tempotools.integer_tempo_to_tempo_multiplier_pairs_report()`. The new name is `tempotools.report_integer_tempo_rewrite_pairs()`.

Removed `durationtools.numeric_seconds_to_escaped_clock_string()`. Use `durationtools.numeric_seconds_to_clock_string(escape_ticks=True)` instead.

Removed `durationtools.is_assignable_rational()`. Use `Duration.is_assignable` property instead.

Removed `durationtools.all_are_duration_tokens()`. Just coerce durations instead.

Removed `durationtools.duration_token_to_duration_pair()`. Just initialize duration objects instead.

Removed `durationtools.is_duration_token()`. Just initialize duration objects instead. Or use `Duration.is_token()` instead if true look-ahead is required.

Removed `durationtools.yield_all_positive_rationals_uniquely()`. Use `durationtools.yield_all_positive_rationals(unique=True)` instead.

Removed `durationtools.assignable_rational_to_dot_count` property. Use `Duration.dot_count` instead.

Removed `durationtools.assignable_rational_to_lilypond_duration_string` property. Use `Duration.lilypond_duration_string` instead.

Removed `durationtools.is_duration_pair()`. Just initialize duration objects instead.

Removed `durationtools.is_binary_rational()`. Use `Duration.is_binary` property instead.

Removed `durationtools.is_proper_tuplet_multiplier()`. Use `Multiplier.is_proper_tuplet_multiplier` property instead.

Removed `durationtools.duration_token_to_rational()`. Just initialize duration objects instead.

Removed `durationtools.duration_tokens_to_rationals()`. Just initialize duration objects instead.

emoved `durationtools.lilypond_duration_string_to_rational()`. Just initialize duration objects instead.

Removed `durationtools.lilypond_duration_string_to_rational_list()`. Function is no longer supported.

Removed `durationtools.rational_to_flag_count()`. Use the `Duration.flag_count` property instead.

Removed `durationtools.rational_to_fraction_string()`. Use `str(Duration)` instead.

Removed `durationtools.rational_to_prolation_string()`. Use the `Duration.prolation_string` property instead.

Renamed `durationtools.rational_to_proper_fraction()`. The new name is `mathtools.fraction_to_proper_fraction()`.

Removed `durationtools.rational_to_duration_pair_with_specified_integer_denominator()`. Use `NonreducedFraction.with_denominator()` instead.

Removed `durationtools.rational_to_duration_pair_with_multiple_of_specified_integer_denomi` Use `mathtools.NonreducedFraction.with_multiple_of_denominator()` instead.

Removed `durationtools.duration_pair_to_prolation_string()`. Use the `Duration.prolation_string` property instead.

Renamed `durationtools.group_duration_tokens_by_implied_prolation()`. The new name is `durationtools.group_nonreduced_fractions_by_implied_prolation()`.

Removed `durationtools.multiply_duration_pair()`. Use `NonreducedFraction.multiply_without_red` instead.

Removed `durationtools.multiply_duration_pair_and_reduce_factors()`. Use `NonreducedFraction.multiply_with_cross_cancelation()` instead.

Removed `durationtools.multiply_duration_pair_and_try_to_preserve_numerator()`. Use `NonreducedFraction.multiply_with_numerator_preservation()` instead.

Removed `durationtools.duration_token_to_assignable_duration_pairs()`. Removed `durationtools.duration_token_to_assignable_rationals()`. Functions are no longer supported. Use `leaftools.make_leaves()` or `notetools.make_notes()` instead.

Removed `durationtools.duration_tokens_to_duration_pairs()`. Function is no longer supported. Just initialize durations intead.

Removed `durationtools.duration_tokens_to_least_common_denominator()`. Function is no longer supported. Use `mathtools.least_common_multiple()` instead.

Renamed `durationtools.duration_tokens_to_duration_pairs_with_least_common_denominator()`. The new name is `durationtools.durations_to_nonreduced_fractions_with_common_denominator()`.

Renamed `durationtools.yield_all_assignable_durations()`. The new name is `durationtools.yield_assignable_durations()`.

Renamed `durationtools.yield_all_positive_integer_pairs()`. The new name is `durationtools.yield_positive_nonreduced_fractions()`.

Renamed `durationtools.yield_all_positive_rationals()`. The new name is `durationtools.yield_positive_fractions()`.

Renamed `durationtools.yield_positive_fractions()`. The new name is `durationtools.yield_durations()`. The function also now returns durations instead of fractions.

Renamed `durationtools.yield_positive_nonreduced_fractions()`. The new name is `durationtools.yield_nonreduced_fractions()`.

Removed `durationtools.yield_prolation_rewrite_pairs()`. The functionality is no longer supported.

Renamed `durationtools.yield_nonreduced_fractions()`. The new name is `mathtools.yield_nonreduced_fractions()`.

Renamed `Duration.is_binary` property. The new name is `Duration.has_power_of_two_denominator`.

Renamed `Measure.is_binary` property. The new name is `Measure.has_power_of_two_denominator`.

Renamed `Tuplet.is_binary` property. The new name is `Tuplet.has_power_of_two_denominator`.

Renamed `Measure.is_nonbinary` property. The new name is `Measure.has_non_power_of_two_denominator`.

Renamed `Tuplet.is_nonbinary` property. The new name is `Tuplet.has_non_power_of_two_denominator`.

Renamed `DynamicMeasure.suppress_meter`. The new name is `DynamicMeasure.suppress_time_signature`.

Removed `durationtools.integer_to_implied_prolation()`. Use the `Duration.implied_prolation` property instead.

Removed unused resttools.is_lilypond_rest_string()‘‘ function. Just instantiate rests instead.

Removed `durationtools.is_lilypond_duration_string()`. Removed `durationtools.is_lilypond_duration_name()`. Just instantiate durations instead.

Removed `componenttools.component_to_score_root()`. Use `Component.parentage.root` instead.

Removed `componenttools.component_to_pitch_and_rhyhtm_skeleton()`. Use the parser instead.

Removed `componenttools.component_to_score_depth()`. Use `Component.parentage.depth` property instead.

Removed unused `componenttools.all_are_orphan_components()` function.

Removed unused `componenttools.all_are_components_in_same_parent()` function.

Removed unused `componenttools.all_are_components_in_same_score()` function.

Removed unused `componenttools.all_are_contiguous_components_in_same_score()` function.

Renamed `leaftools.make_leaves_from_note_value_signal()`. The new name is `leaftools.make_leaves_from_talea()`.

Removed `TimeSignatureMark.multiplier` property. Use `TimeSignatureMark.implied_prolation` instead.

Removed `Measure.multiplier` property. Use `Measure.implied_prolation` instead.

Deprecated `timesignaturetools.time_signature_to_time_signature_with_power_of_two_denominato` function. Use `TimeSignatureMark.with_power_of_two_denominator()` method instead.

Remvoed `timesignaturetools.time_signature_to_time_signature_with_power_of_two_denominator` function. Use `TimeSignatureMark.with_power_of_two_denominator()` method instead.

Moved one function from `componenttools` to `measuretools`. The function is `get_likely_multiplier_components()`.

Moved one function from `componenttools` to `formattools`. The function is `report_component_format_contributions()`.

Globally replaced rhythm maker `pattern` names to `talea`. The name change harmonizes with the new names fo the rhythm maker classes.

Removed `big endian` and `little endian` from codebase. Use `decrease_durations_monotonically=True` keyword instead.

Removed the word `duration_token` from mainline. The term is deprecated. Use `duration` instead.

Deprecated the term `pitch_token`. Use `pitch` instead.

Removed `pitchtools.named_chromatic_pitch_tokens_to_named_chromatic_pitches()`. Just instantiate pitches instead.

Removed the term `signal` from the `rhythmmakertools` package. Use `talea` instead. The plural of `talea` is `talee`.

Moved `componenttools.component_to_tuplet_depth()`. The function is now bound to parentage as the `Component.parentage.tuplet_depth` property.

Moved `componenttools.component_to_score_index()`. The function is now bound to parentage as the `Component.parentage.score_index` property.

Moved `componenttools.component_to_containment_signature()`. The function is now bound to parentage as the `Component.parentage.containment_signature` property.

Moved `componenttools.component_to_parentage_signature()`. The function is now bound to parentage as the `Component.parentage.parentage_signature` property.

Renamed `componenttools.cut_component_by_at_prolated_duration()`. The new name is `componenttools.shorten_component_by_prolated_duration()`.

Renamed `componenttools.get_leftmost_components_with_prolated_duration_at_most()`. The new name is `componenttools.get_leftmost_components_with_total_duration_at_most()`.

Renamed `componenttools.shorten_component_by_prolated_duration()`. The new name is `componenttools.shorten_component_by_duration()`.

Renamed `componenttools.sum_prolated_duration_of_components()`. The new name is `componenttools.sum_duration_of_components()`.

Renamed `componenttools.yield_components_grouped_by_prolated_duration()`. The new name is `componenttools.yield_components_grouped_by_duration()`.

Renamed `labeltools.label_leaves_in_expr_with_prolated_leaf_duration()`. The new name is `labeltools.label_leaves_in_expr_with_leaf_duration()`.

Renamed `labeltools.label_tie_chains_in_expr_with_prolated_tie_chain_duration()`. The new name is `labeltools.label_tie_chains_in_expr_with_tie_chain_duration()`.

Renamed `leaftools.fuse_tied_leaves_in_components_once_by_prolated_durations_without_overh`
The new name is `leaftools.fuse_tied_leaves_in_components_once_by_durations_without_overhang(`

Renamed `leaftools.get_leaf_in_expr_with_maximum_prolated_duration()`. The new name is `leaftools.get_leaf_in_expr_with_maximum_duration()`.

Renamed `leaftools.get_leaf_in_expr_with_minimum_prolated_duration()`. The new name is `leaftools.get_leaf_in_expr_with_minimum_duration()`.

Rename `leaftools.list_prolated_durations_of_leaves_in_expr()`. The new name is `leaftools.list_durations_of_leaves_in_expr()`.

Renamed `VerticalMoment.prolated_offset` to `VerticalMoment.offset`.

Merged `componenttools.extend_left_in_parent_of_component()` into `componenttools.extend_in_parent_of_component()`. Use the `left=True` keyword.

Removed `componenttools.extend_left_in_parent_of_component()` Use `componenttools.extend_in_parent_of_component(left=True)` instead.

Removed `componenttools.get_component_start_offset()`. Removed `componenttools.get_component_stop_offset()`. Use the `Component.start_offset` and `Component.stop_offset` properties instead.

Removed `componenttools.get_component_start_offset_in_seconds()`. Removed `componenttools.get_component_stop_offset_in_seconds()`. Use the `Component.start_offset_in_seconds` and `Component.stop_offset_in_seconds` properties instead.

Removed `componenttools.is_orphan_component()`. Use the new `Component.parentage.is_orphan` property instead.

Renamed `componenttools.partition_components_by_durations_ge()` The new name is `componenttools.partition_components_by_durations_not_less_than()`

Renamed `componenttools.partition_components_by_durations_le()` The new name is `componenttools.partition_components_by_durations_not_greater_than()`

emoved `componenttools.sum_preprolated_duration_of_components()` Use `componenttools.sum_duration_of_components(preprolated=True)` instead.

Removed `componenttools.sum_duration_of_components_in_seconds()`. Use `componenttools.sum_duration_of_components(in_seconds=True)` instead.

Changed ratio objects to reduce terms at initialization.

Changed `diminution` keyword to `is_diminution` in three functions:

```
tuplettools.leaf_to_tuplet_with_proportions()
tuplettools.leaf_to_tuplet_with_n_notes_of_equal_written_duration()
tietools.tie_chain_to_tuplet_with_proportions()
```

Moved three functions from `componenttools` to `wellformednesstools`. The functions are these:

```
is_well_formed_component()
list_badly_formed_components_in_expr()
tabulate_well_formedness_violations_in_expr()
```

Removed two `componenttools` functions. Use `timerelationtools` instead. The functions are these:

```
componenttools.number_is_between_start_and_stop_offsets_of_component()
componenttools.number_is_between_start_and_stop_offsets_of_component_in_seconds()
```

Renamed `tied=True` keyword in four functions:

```
leaftools.make_leaves()
leaftools.make_tied_leaf()
resttools.make_tied_rest()
resttools.make_rests()
```

Renamed the four ratio-related API functions:

```
tietools.tie_chain_to_tuplet_with_proportions()
tuplettools.leaf_to_tuplet_with_proportions()
tuplettools.make_tuplet_from_duration_and_proportions()
tuplettools.make_tuplet_from_proportions_and_pair()
```

```
tietools.tie_chain_to_tuplet_with_ratio()
tuplettools.leaf_to_tuplet_with_ratio()
tuplettools.make_tuplet_from_duration_and_ratio()
tuplettools.make_tuplet_from_nonreduced_ratio_and_nonreduced_fraction()
```

Added four new public properties to `Duration` that replace functions:

```
Duration.equal_or_greater_assignable
Duration.equal_or_greater_power_of_two
Duration.equal_or_lesser_assignable
Duration.equal_or_lesser_power_of_two
```

```
durationtools.rational_to_equal_or_greater_assignable_rational()
durationtools.rational_to_equal_or_greater_binary_rational()
durationtools.rational_to_equal_or_lesser_assignable_rational()
durationtools.rational_to_equal_or_lesser_binary_rational()
```

63.2 Older Versions

63.2.1 Changes from 2.9 to 2.10

Renamed the read-only `format` property to `lilypond_format` on all system objects.

All iteration functions are now housed in the new `iterationtools` package:

- Renamed:

```
chordtools.iterate_chords_forward_in_expr()
chordtools.iterate_chords_backward_in_expr()
```

```
iterationtools.iterate_chords_in_expr(reverse=[True, False])
```

- Renamed:

```
componenttools.iterate_components_depth_first()
componenttools.iterate_components_forward_in_expr()
componenttools.iterate_components_backward_in_expr()
componenttools.iterate_namesakes_forward_from_component()
componenttools.iterate_namesakes_backward_from_component()
componenttools.iterate_thread_forward_from_component()
componenttools.iterate_thread_backward_from_component()
componenttools.iterate_thread_forward_in_expr()
componenttools.iterate_thread_backward_in_expr()
componenttools.iterate_timeline_forward_from_component()
componenttools.iterate_timeline_backward_from_component()
componenttools.iterate_timeline_forward_in_expr()
componenttools.iterate_timeline_backward_in_expr()
```

```
iterationtools.iterate_components_depth_first()
iterationtools.iterate_components_in_expr(reverse=[True, False])
iterationtools.iterate_namesakes_from_component(reverse=[True, False])
iterationtools.iterate_thread_from_component(reverse=[True, False])
iterationtools.iterate_thread_in_expr(reverse=[True, False])
iterationtools.iterate_timeline_from_component(reverse=[True, False])
iterationtools.iterate_timeline_in_expr(reverse=[True, False])
```

- Renamed:

```
containertools.iterate_containers_forward_in_expr()
containertools.iterate_containers_backward_in_expr()
```

```
iterationtools.iterate_containers_in_expr(reverse=[True, False])
```

- Renamed:

```
contexttools.iterate_contexts_forward_in_expr()
contexttools.iterate_contexts_backward_in_expr()
```

```
iterationtools.iterate_contexts_in_expr(reverse=[True, False])
```

- Renamed:

```
gracetools.iterate_components_and_grace_containers_forward_in_expr()
```

```
iterationtools.iterate_components_and_grace_containers_in_expr()
```

- Renamed:

```
leaftools.iterate_leaf_pairs_forward_in_expr()
leaftools.iterate_leaves_forward_in_expr()
leaftools.iterate_leaves_backward_in_expr()
leaftools.iterate_notes_and_chords_forward_in_expr()
leaftools.iterate_notes_and_chords_backward_in_expr()
```

```
iterationtools.iterate_leaf_pairs_in_expr()
iterationtools.iterate_leaves_in_expr(reverse=[True, False])
iterationtools.iterate_notes_and_chords_in_expr(reverse=[True, False])
```

- Renamed:

```
measuretools.iterate_measures_forward_in_expr()
measuretools.iterate_measures_backward_in_expr()
```

```
iterationtools.iterate_measures_in_expr(reverse=[True, False])
```

- Renamed:

```
notetools.iterate_notes_forward_in_expr()
notetools.iterate_notes_backward_in_expr()
```

```
iterationtools.iterate_notes_in_expr(reverse=[True, False])
```

- Renamed:

```
resttools.iterate_rests_forward_in_expr()
resttools.iterate_rests_backward_in_expr()
```

```
iterationtools.iterate_rests_in_expr(reverse=[True, False])
```

- Renamed:

```
scoretools.iterate_scores_forward_in_expr()
scoretools.iterate_scores_backward_in_expr()
```

```
iterationtools.iterate_scores_in_expr(reverse=[True, False])
```

- Renamed:

```
skiptools.iterate_skips_forward_in_expr()
skiptools.iterate_skips_backward_in_expr()
```

```
iterationtools.iterate_skips_in_expr(reverse=[True, False])
```

- Renamed:

```
stafftools.iterate_staves_forward_in_expr()
stafftools.iterate_staves_backward_in_expr()
```

```
iterationtools.iterate_staves_in_expr(reverse=[True, False])
```

- Renamed:

```
tuplettools.iterate_tuplets_forward_in_expr()
tuplettools.iterate_tuplets_backward_in_expr()
```

```
iterationtools.iterate_tuplets_in_expr(reverse=[True, False])
```

- Renamed:

```
voicetools.iterate_semantic_voices_forward_in_expr()
voicetools.iterate_semantic_voices_backward_in_expr()
voicetools.iterate_voices_forward_in_expr()
voicetools.iterate_voices_backward_in_expr()
```

```
voicetools.iterate_semantic_voices_in_expr(reverse=[True, False])
voicetools.iterate_voices_in_expr(reverse=[True, False])
```

All labeling functions are now housed in the new `labeltools` package:

- Renamed:

```
chordtools.color_chord_note_heads_in_expr_by_pitch_class_color_map()
```

```
labeltools.color_chord_note_heads_in_expr_by_pitch_class_color_map()
```

- Renamed:

```
containertools.color_contents_of_container()
```

```
labeltools.color_contents_of_container()
```

- Renamed:

```
leaftools.color_leaf()
leaftools.color_leaves_in_expr()
leaftools.label_leaves_in_expr_with_inversion_equivalent_chromatic_interval_classes()
leaftools.label_leaves_in_expr_with_leaf_depth()
leaftools.label_leaves_in_expr_with_leaf_durations()
leaftools.label_leaves_in_expr_with_leaf_indices()
leaftools.label_leaves_in_expr_with_leaf_numbers()
leaftools.label_leaves_in_expr_with_melodic_chromatic_interval_classes()
leaftools.label_leaves_in_expr_with_melodic_chromatic_intervals()
leaftools.label_leaves_in_expr_with_melodic_counterpoint_interval_classes()
leaftools.label_leaves_in_expr_with_melodic_counterpoint_intervals()
leaftools.label_leaves_in_expr_with_melodic_diatonic_interval_classes()
leaftools.label_leaves_in_expr_with_melodic_diatonic_intervals()
leaftools.label_leaves_in_expr_with_pitch_class_numbers()
leaftools.label_leaves_in_expr_with_pitch_numbers()
leaftools.label_leaves_in_expr_with_leaf_duration()
leaftools.label_leaves_in_expr_with_tuplet_depth()
leaftools.label_leaves_in_expr_with_written_leaf_duration()
```

```
labeltools.color_leaf()
labeltools.color_leaves_in_expr()
labeltools.label_leaves_in_expr_with_inversion_equivalent_chromatic_interval_classes()
labeltools.label_leaves_in_expr_with_leaf_depth()
labeltools.label_leaves_in_expr_with_leaf_durations()
labeltools.label_leaves_in_expr_with_leaf_indices()
labeltools.label_leaves_in_expr_with_leaf_numbers()
labeltools.label_leaves_in_expr_with_melodic_chromatic_interval_classes()
labeltools.label_leaves_in_expr_with_melodic_chromatic_intervals()
labeltools.label_leaves_in_expr_with_melodic_counterpoint_interval_classes()
labeltools.label_leaves_in_expr_with_melodic_counterpoint_intervals()
labeltools.label_leaves_in_expr_with_melodic_diatonic_interval_classes()
labeltools.label_leaves_in_expr_with_melodic_diatonic_intervals()
labeltools.label_leaves_in_expr_with_pitch_class_numbers()
labeltools.label_leaves_in_expr_with_pitch_numbers()
labeltools.label_leaves_in_expr_with_leaf_duration()
labeltools.label_leaves_in_expr_with_tuplet_depth()
labeltools.label_leaves_in_expr_with_written_leaf_duration()
```

- Renamed:

```
markuptools.remove_markup_from_leaves_in_expr()
```

```
labeltools.remove_markup_from_leaves_in_expr()
```

- Renamed:

```
measuretools.color_measure()
measuretools.color_measures_with_non_power_of_two_denominators_in_expr()
```

```
labeltools.color_measure()
labeltools.color_measures_with_non_power_of_two_denominators_in_expr()
```

- Renamed:

```
notetools.color_note_head_by_numbered_chromatic_pitch_class_color_map()
notetools.label_notes_in_expr_with_note_indices()
```

```
labeltools.color_note_head_by_numbered_chromatic_pitch_class_color_map()
labeltools.label_notes_in_expr_with_note_indices()
```

- Renamed:

```
tietools.label_tie_chains_in_expr_with_tie_chain_duration()
tietools.label_tie_chains_in_expr_with_tie_chain_durations()
tietools.label_tie_chains_in_expr_with_written_tie_chain_duration()
```

```
labeltools.label_tie_chains_in_expr_with_tie_chain_duration()
labeltools.label_tie_chains_in_expr_with_tie_chain_durations()
labeltools.label_tie_chains_in_expr_with_written_tie_chain_duration()
```

- Renamed:

```
verticalitytools.label_vertical_moments_in_expr_with_chromatic_interval_classes()
verticalitytools.label_vertical_moments_in_expr_with_chromatic_intervals()
verticalitytools.label_vertical_moments_in_expr_with_counterpoint_intervals()
verticalitytools.label_vertical_moments_in_expr_with_diatonic_intervals()
verticalitytools.label_vertical_moments_in_expr_with_interval_class_vectors()
verticalitytools.label_vertical_moments_in_expr_with_numbered_chromatic_pitch_classes()
verticalitytools.label_vertical_moments_in_expr_with_pitch_numbers()
```

```
labeltools.label_vertical_moments_in_expr_with_chromatic_interval_classes()
labeltools.label_vertical_moments_in_expr_with_chromatic_intervals()
labeltools.label_vertical_moments_in_expr_with_counterpoint_intervals()
labeltools.label_vertical_moments_in_expr_with_diatonic_intervals()
labeltools.label_vertical_moments_in_expr_with_interval_class_vectors()
labeltools.label_vertical_moments_in_expr_with_numbered_chromatic_pitch_classes()
labeltools.label_vertical_moments_in_expr_with_pitch_numbers()
```

Renamed all functions that contained `big_endian`:

```
durationtools.duration_token_to_big_endian_list_of_assignable_duration_pairs()
leaftools.fuse_leaves_big_endian()
leaftools.fuse_leaves_in_tie_chain_by_immediate_parent_big_endian()
```

```
durationtools.duration_token_to_assignable_duration_pairs()
leaftools.fuse_leaves()
leaftools.fuse_leaves_in_tie_chain_by_immediate_parent()
```

Renamed all functions that contained `prolated_offset` to simply `offset`:

```
componenttools.copy_governed_component_subtree_from_prolated_offset_to()
componenttools.get_improper_descendents_of_component_that_cross_prolated_offset()
containertools.delete_contents_of_container_starting_at_or_after_prolated_offset()
containertools.delete_contents_of_container_starting_before_or_at_prolated_offset()
containertools.delete_contents_of_container_starting_strictly_after_prolated_offset()
containertools.delete_contents_of_container_starting_strictly_before_prolated_offset()
containertools.get_element_starting_at_exactly_prolated_offset()
containertools.get_first_element_starting_at_or_after_prolated_offset()
containertools.get_first_element_starting_before_or_at_prolated_offset()
containertools.get_first_element_starting_strictly_after_prolated_offset()
containertools.get_first_element_starting_strictly_before_prolated_offset()
prolated_offsettools.update_offset_values_of_component()
verticalitytools.get_vertical_moment_at_prolated_offset_in_expr()
```

```
componenttools.copy_governed_component_subtree_from_offset_to()
componenttools.get_improper_descendents_of_component_that_cross_offset()
containertools.delete_contents_of_container_starting_at_or_after_offset()
containertools.delete_contents_of_container_starting_before_or_at_offset()
containertools.delete_contents_of_container_starting_strictly_after_offset()
containertools.delete_contents_of_container_starting_strictly_before_offset()
containertools.get_element_starting_at_exactly_offset()
containertools.get_first_element_starting_at_or_after_offset()
containertools.get_first_element_starting_before_or_at_offset()
containertools.get_first_element_starting_strictly_after_offset()
containertools.get_first_element_starting_strictly_before_offset()
offsettools.update_offset_values_of_component()
verticalitytools.get_vertical_moment_at_offset_in_expr()
```

Renamed `prolated_duration` to `offset` in some functions:

```
componenttools.split_component_at_prolated_duration()
componenttools.split_components_by_prolated_durations()
leaftools.split_leaf_at_prolated_duration()
leaftools.split_leaf_at_prolated_duration_and_rest_right_half()
```

```
componenttools.split_component_at_offset()
componenttools.split_components_by_offsets()
leaftools.split_leaf_at_offset()
leaftools.split_leaf_at_offset_and_rest_right_half()
```

Renamed all functions that contained `as_string`:

```
componenttools.report_component_format_contributions_as_string()
containertools.report_container_modifications_as_string()
measuretools.report_meter_distribution_as_string()
```

```
formattools.report_component_format_contributions()
containertools.report_container_modifications()
measuretools.report_time_signature_distribution()
```

Changes to the `componenttools` package:

- The `componenttools.split_components_at_offsets()` function no longer implements a `tie_after keyword`. Use the new `tie_split_notes` and `tie_split_rests` keywords. Note that the new `tie_split_rests` keyword defaults to true where the old `tie_after` keyword defaulted to false. This changes the default behavior of the function.
- Renamed:

```
componenttools.extend_left_in_parent_of_component_and_grow_spanners()
componenttools.extend_left_in_parent_of_component_and_do_not_grow_spanners()
```

```
componenttools.extend_left_in_parent_of_component(grow_spanners=[True, False])
```

- Renamed:

```
componenttools.extend_in_parent_of_component_and_grow_spanners()
componenttools.extend_in_parent_of_component_and_do_not_grow_spanners()
```

```
componenttools.extend_in_parent_of_component(grow_spanners=[True, False])
```

- Renamed:

```
componenttools.number_is_between_prolated_start_and_stop_offsets_of_component()
```

```
componenttools.number_is_between_start_and_stop_offsets_of_component()
```

- Renamed:

```
componenttools.partition_components_cyclically_by_durations_in_seconds_exactly_with_overhang()
componenttools.partition_components_cyclically_by_durations_in_seconds_exactly_without_overhang()
componenttools.partition_components_cyclically_by_durations_in_seconds_ge_with_overhang()
componenttools.partition_components_cyclically_by_durations_in_seconds_ge_without_overhang()
componenttools.partition_components_cyclically_by_durations_in_seconds_le_with_overhang()
componenttools.partition_components_cyclically_by_durations_in_seconds_le_without_overhang()
componenttools.partition_components_cyclically_by_prolated_durations_exactly_with_overhang()
componenttools.partition_components_cyclically_by_prolated_durations_exactly_without_overhang()
componenttools.partition_components_cyclically_by_prolated_durations_ge_with_overhang()
componenttools.partition_components_cyclically_by_prolated_durations_ge_without_overhang()
componenttools.partition_components_cyclically_by_prolated_durations_le_with_overhang()
componenttools.partition_components_cyclically_by_prolated_durations_le_without_overhang()
componenttools.partition_components_once_by_durations_in_seconds_exactly_with_overhang()
componenttools.partition_components_once_by_durations_in_seconds_exactly_without_overhang()
componenttools.partition_components_once_by_durations_in_seconds_ge_with_overhang()
componenttools.partition_components_once_by_durations_in_seconds_ge_without_overhang()
componenttools.partition_components_once_by_durations_in_seconds_le_with_overhang()
componenttools.partition_components_once_by_durations_in_seconds_le_without_overhang()
componenttools.partition_components_once_by_prolated_durations_exactly_with_overhang()
```

```
componenttools.partition_components_once_by_prolated_durations_exactly_without_overhang()
componenttools.partition_components_once_by_prolated_durations_ge_with_overhang()
componenttools.partition_components_once_by_prolated_durations_ge_without_overhang()
componenttools.partition_components_once_by_prolated_durations_le_with_overhang()
componenttools.partition_components_once_by_prolated_durations_le_without_overhang()
```

```
componenttools.partition_components_by_durations_exactly()
componenttools.partition_components_by_durations_not_less_than()
componenttools.partition_components_by_durations_not_greater_than()
```

- Renamed:

```
componenttools.split_component_at_prolated_duration_and_do_not_fracture_crossing_spanners()
componenttools.split_component_at_prolated_duration_and_fracture_crossing_spanners()
```

```
componenttools.split_component_at_offset(fracture_spanners=[True, False])
```

- Renamed:

```
componenttools.split_components_cyclically_by_prolated_durations_and_do_not_fracture_crossing_spanners()
componenttools.split_components_cyclically_by_prolated_durations_and_fracture_crossing_spanners()
componenttools.split_components_once_by_prolated_durations_and_do_not_fracture_crossing_spanners()
componenttools.split_components_once_by_prolated_durations_and_fracture_crossing_spanners()
```

```
componenttools.split_components_at_offsets(fracture_spanners=[True, False], cyclic=[True, False])
```

Changeds to the `continertools` package:

- Renamed:

```
containertools.remove_empty_containers_in_expr()
```

```
containertools.remove_leafless_containers_in_expr()
```

- Renamed:

```
containertools.replace_larger_left_half_of_elements_in_container_with_big_endian_rests()
containertools.replace_larger_left_half_of_elements_in_container_with_little_endian_rests()
containertools.replace_larger_right_half_of_elements_in_container_with_big_endian_rests()
containertools.replace_larger_right_half_of_elements_in_container_with_little_endian_rests()
containertools.replace_n_edge_elements_in_container_with_big_endian_rests()
containertools.replace_n_edge_elements_in_container_with_little_endian_rests()
containertools.replace_n_edge_elements_in_container_with_rests()
containertools.replace_smaller_left_half_of_elements_in_container_with_big_endian_rests()
containertools.replace_smaller_left_half_of_elements_in_container_with_little_endian_rests()
containertools.replace_smaller_right_half_of_elements_in_container_with_big_endian_rests()
containertools.replace_smaller_right_half_of_elements_in_container_with_little_endian_rests()
```

```
containertools.replace_container_slice_with_rests()
```

- Renamed:

```
containertools.split_container_at_index_and_do_not_fracture_crossing_spanners()
containertools.split_container_at_index_and_fracture_crossing_spanners()
```

```
containertools.split_container_at_index(fracture_spanners=[True, False])
```

- Renamed:

```
containertools.split_container_cyclically_by_counts_and_do_not_fracture_crossing_spanners()
containertools.split_container_cyclically_by_counts_and_fracture_crossing_spanners()
containertools.split_container_once_by_counts_and_do_not_fracture_crossing_spanners()
containertools.split_container_once_by_counts_and_fracture_crossing_spanners()
```

```
containertools.split_container_by_counts(fracture_spanners=[True, False], cyclic=[True, False])
```

Changes to the `durationtools` package:

- Renamed:

```
    durationtools.yield_all_assignable_rationals_in_cantor_diagonalized_order()
    durationtools.yield_all_positive_integer_pairs_in_cantor_diagonalized_order()
    durationtools.yield_all_positive_rationals_in_cantor_diagonalized_order()
    durationtools.yield_all_positive_rationals_in_cantor_diagonalized_order_uniquely()
    durationtools.yield_all_prolation_rewrite_pairs_of_rational_in_cantor_diagonalized_order()
```

```
    durationtools.yield_assignable_durations()
    mathtools.yield_nonreduced_fractions()
    durationtools.yield_durations()
    durationtools.yield_all_positive_rationals_uniquely()
    metricmodulationtools.yield_prolation_rewrite_pairs()
```

Changes to the `instrumenttools` package:

- Renamed:

```
    instrumenttools.transpose_notes_and_chords_in_expr_from_sounding_pitch_to_fingered_pitch()
```

```
    instrumenttools.transpose_from_sounding_pitch_to_fingered_pitch()
```

- Renamed:

```
    instrumenttools.transpose_notes_and_chords_in_expr_from_fingered_pitch_to_sounding_pitch()
```

```
    instrumenttools.transpose_from_fingered_pitch_to_sounding_pitch()
```

Chnages to the `leaftools` package:

- Renamed:

```
    leaftools.fuse_leaves_in_container_once_by_counts_into_big_endian_notes()
    leaftools.fuse_leaves_in_container_once_by_counts_into_big_endian_rests()
    leaftools.fuse_leaves_in_container_once_by_counts_into_little_endian_notes()
    leaftools.fuse_leaves_in_container_once_by_counts_into_little_endian_rests()
```

```
    leaftools.fuse_leaves_in_container_once_by_counts(big_endian=[True, False], klass=None)
```

- Renamed:

```
    leaftools.leaf_to_augmented_tuplet_with_n_notes_of_equal_written_duration()
    leaftools.leaf_to_augmented_tuplet_with_proportions()
    leaftools.leaf_to_diminished_tuplet_with_n_notes_of_equal_written_duration()
    leaftools.leaf_to_diminished_tuplet_with_proportions()
```

```
    tuplettools.leaf_to_tuplet_with_n_notes_of_equal_written_duration()
    tuplettools.leaf_to_tuplet_with_ratio()
```

- Renamed:

```
    leaftools.split_leaf_at_offset_and_rest_right_half()
```

```
    leaftools.rest_leaf_at_offset()
```

- Renamed:

```
    leaftools.repeat_leaf_and_extend_spanners()
    leaftools.repeat_leaves_in_expr_and_extend_spanners()
```

```
    leaftools.repeat_leaf()
    leaftools.repeat_leaves_in_expr()
```

Changes to the `mathtools` package.

- Removed `mathtools.partition_integer_into_thirds()`.

Changes to the `measuretools` package:

- Renamed:

```
measuretools.fill_measures_in_expr_with_meter_denominator_notes()
measuretools.move_prolation_of_full_measure_tuplet_to_meter_of_measure()
measuretools.multiply_contents_of_measures_in_expr_and_scale_meter_denominators()
measuretools.scale_measure_by_multiplier_and_adjust_meter()
```

```
measuretools.fill_measures_in_expr_with_time_signature_denominator_notes()
measuretools.move_full_measure_tuplet_prolation_to_measure_time_signature()
measuretools.multiply_contents_of_measures_in_expr_and_scale_time_signature_denominators()
measuretools.scale_measure_and_adjust_time_signature()
```

- Renamed:

```
measuretools.fill_measures_in_expr_with_big_endian_notes()
measuretools.fill_measures_in_expr_with_litte_endian_notes()
```

```
measuretools.measuretools.fill_measures_in_expr_with_minimal_number_of_notes(big_endian=[True, False])
```

- Renamed:

```
measuretools.extend_measures_in_expr_and_apply_full_measure_tuplets_to_measure_contents()
```

```
measuretoools.extend_measures_in_expr_and_apply_full_measure_tuplets()
```

- Renamed:

```
measuretools.get_previous_measure_from_component()
```

```
measuretools.get_previous_measure_from_component()
```

- Renamed:

```
measuretools.multiply_contents_of_measures_in_expr_and_scale_time_signature_denominators()
```

```
measuretools.multiply_and_scale_contents_of_measures_in_expr()
```

- Renamed:

```
measuretools.pitch_array_row_to_measure()
measuretools.pitch_array_to_measures()
```

```
pitchtools.pitch_array_row_to_measure()
pitchtools.pitch_array_to_measures()
```

Changes to the `pitchtools` package:

- Renamed:

```
pitchtools.calculate_harmonic_chromatic_interval_class_from_pitch_carrier_to_pitch_carrier()
pitchtools.calculate_harmonic_chromatic_interval_from_pitch_carrier_to_pitch_carrier()
pitchtools.calculate_harmonic_counterpoint_interval_class_from_named_chromatic_pitch_to_named_chromatic_p
pitchtools.calculate_harmonic_counterpoint_interval_from_named_chromatic_pitch_to_named_chromatic_pitch()
pitchtools.calculate_harmonic_diatonic_interval_class_from_named_chromatic_pitch_to_named_chromatic_pitch
pitchtools.calculate_harmonic_diatonic_interval_from_named_chromatic_pitch_to_named_chromatic_pitch()
```

```
pitchtools.calculate_harmonic_chromatic_interval_class()
pitchtools.calculate_harmonic_chromatic_interval()
pitchtools.calculate_harmonic_counterpoint_interval_class()
pitchtools.calculate_harmonic_counterpoint_interval()
pitchtools.calculate_harmonic_diatonic_interval_class()
pitchtools.calculate_harmonic_diatonic_interval()
```

- Renamed:

```
pitchtools.calculate_melodic_chromatic_interval_class_from_pitch_carrier_to_pitch_carrier()
pitchtools.calculate_melodic_chromatic_interval_from_pitch_carrier_to_pitch_carrier()
pitchtools.calculate_melodic_counterpoint_interval_class_from_named_chromatic_pitch_to_named_chromatic_p
pitchtools.calculate_melodic_counterpoint_interval_from_named_chromatic_pitch_to_named_chromatic_pitch()
pitchtools.calculate_melodic_diatonic_interval_class_from_named_chromatic_pitch_to_named_chromatic_pitch
pitchtools.calculate_melodic_diatonic_interval_from_named_chromatic_pitch_to_named_chromatic_pitch()
```

```
pitchtools.calculate_melodic_chromatic_interval_class()
pitchtools.calculate_melodic_chromatic_interval()
pitchtools.calculate_melodic_counterpoint_interval_class()
pitchtools.calculate_melodic_counterpoint_interval()
pitchtools.calculate_melodic_diatonic_interval_class()
pitchtools.calculate_melodic_diatonic_interval()
```

- Renamed:

```
pitchtools.chromatic_pitch_class_name_to_diatonic_pitch_class_name_alphabetic_accidental_abbreviation_pa
```

```
pitchtools.split_chromatic_pitch_class_name()
```

- Renamed:

```
pitchtools.diatonic_interval_number_and_chromatic_interval_number_to_melodic_diatonic_interval()
```

```
pitchtools.spell_chromatic_interval_number()
```

- Renamed:

```
pitchtools.named_chromatic_pitches_to_harmonic_chromatic_interval_class_number_dictionary()
```

```
pitchtools.harmonic_chromatic_interval_class_number_dictionary()
```

- Renamed:

```
pitchtools.chromatic_pitch_number_diatonic_pitch_class_name_to_alphabetic_accidental_abbreviation_octave
```

```
pitchtools.chromatic_pitch_number_diatonic_pitch_class_name_to_accidental_octave_number_pair()
```

- Renamed:

```
pitchtools.list_named_chromatic_pitch_carriers_in_expr_sorted_by_numbered_chromatic_pitch_class()
```

```
pitchtools.sort_named_chromatic_pitch_carriers_in_expr()
```

- Renamed:

```
pitchtools.named_chromatic_pitches_to_inversion_equivalent_chromatic_interval_class_number_dictionary()
```

```
pitchtools.inversion_equivalent_chromatic_interval_class_number_dictionary()
```

- Renamed:

```
pitchtools.transpose_chromatic_pitch_class_number_by_octaves_to_nearest_neighbor_of_chromatic_pitch_numb
```

```
pitchtools.transpose_chromatic_pitch_class_number_to_neighbor_of_chromatic_pitch_number()
```

- Renamed:

```
pitchtools.ordered_chromatic_pitch_class_numbers_are_within_ordered_chromatic_pitch_numbers()
```

```
pitchtools.contains_subsegment()
```

- Renamed:

```
pitchtools.list_inversion_equivalent_chromatic_interval_classes_pairwise_between_pitch_carriers()
```

```
pitchtools.list_inversion_equivalent_chromatic_interval_classes_pairwise()
```

- Renamed:

```
pitchtools.list_melodic_chromatic_interval_numbers_pairwise_between_pitch_carriers()
```

```
pitchtools.list_melodic_chromatic_interval_numbers_pairwise()
```

- Renamed:

```
pitchtools.chromatic_pitch_number_to_diatonic_pitch_class_name_accidental_octave_number_triple()
```

```
pitchtools.chromatic_pitch_number_to_chromatic_pitch_triple()
```

- Renamed:

```
pitchtools.apply_octavation_spanner_to_pitched_components()
```

```
spannertools.apply_octavation_spanner_to_pitched_components()
```

- Renamed:

```
pitchtools.set_ascending_named_chromatic_pitches_on_nontied_pitched_components_in_expr()
```

```
pitchtools.set_ascending_named_chromatic_pitches_on_tie_chains_in_expr()
```

- Renamed:

```
pitchtools.set_ascending_diatonic_pitches_on_nontied_pitched_components_in_expr()
```

```
pitchtools.set_ascending_diatonic_pitches_on_tie_chains_in_expr()
```

- Renamed:

```
pitchtools.transpose_chromatic_pitch_class_number_to_neighbor_of_chromatic_pitch_number()
```

```
pitchtools.transpose_chromatic_pitch_class_number_chromatic_pitch_number_neighbor()
```

Changes to the `rhythmtreetools` package:

- Renamed:

```
rhythmtreetools.parse_reduced_ly_syntax()
```

```
lilypondparsertools.parse_reduced_ly_syntax()
```

Chnages to the `scoretemplatetools` package:

- Renamed:

```
scoretemplatetools.GroupedRhythmcStavesScoreTemplate.n
```

```
scoretemplatetools.GroupedRhythmcStavesScoreTemplate.staff_count
```

Changes to the `scoretools` package:

- Renamed:

```
scoretools.make_pitch_array_score_from_pitch_arrays()
```

```
pitchtools.make_pitch_array_score_from_pitch_arrays()
```

Changes to the `sequencetools` package:

- Renamed:

```
sequencetools.partition_sequence_cyclically_by_counts_with_overhang()
sequencetools.partition_sequence_cyclically_by_counts_without_overhang()
sequencetools.partition_sequence_once_by_counts_with_overhang()
sequencetools.partition_sequence_once_by_counts_without_overhang()
```

```
sequencetools.partition_sequence_by_counts(cyclic=[True, False], overhang=[True, False])
```

- Renamed:

```
sequencetools.partition_sequence_extended_to_counts_with_overhang()
sequencetools.partition_sequence_extended_to_counts_without_overhang()
```

```
sequencetools.partition_sequence_extended_to_counts(overhang=[True, False])
```

- Renamed:

```
sequencetools.partition_sequence_cyclically_by_weights_at_least_with_overhang()
sequencetools.partition_sequence_cyclically_by_weights_at_least_without_overhang()
sequencetools.partition_sequence_once_by_weights_at_least_with_overhang()
sequencetools.partition_sequence_once_by_weights_at_least_without_overhang()
```

```
sequencetools.partition_sequence_by_weights_at_least()
```

- Renamed:

```
sequencetools.partition_sequence_cyclically_by_weights_at_most_with_overhang()
sequencetools.partition_sequence_cyclically_by_weights_at_most_without_overhang()
sequencetools.partition_sequence_once_by_weights_at_most_with_overhang()
sequencetools.partition_sequence_once_by_weights_at_most_without_overhang()
```

```
sequencetools.partition_sequence_by_weights_at_most()
```

- Renamed:

```
sequencetools.partition_sequence_cyclically_by_weights_at_exactly_with_overhang()
sequencetools.partition_sequence_cyclically_by_weights_at_exactly_without_overhang()
sequencetools.partition_sequence_once_by_weights_at_exactly_with_overhang()
sequencetools.partition_sequence_once_by_weights_at_exactly_without_overhang()
```

```
sequencetools.partition_sequence_by_weights_at_exactly()
```

- Renamed:

```
sequencetools.split_sequence_cyclically_by_weights_with_overhang()
sequencetools.split_sequence_cyclically_by_weights_without_overhang()
sequencetools.split_sequence_once_by_weights_with_overhang()
sequencetools.split_sequence_once_by_weights_without_overhang()
```

```
sequencetools.split_sequence_by_weights()
```

- Renamed:

```
sequencetools.split_sequence_extended_to_weights_with_overhang()
sequencetools.split_sequence_extended_to_weights_without_overhang()
```

```
sequencetools.split_sequence_extended_to_weights()
```

Changes to the `tietools` package:

- Renamed:

```
tietools.tie_chain_to_augmented_tuplet_with_proportions_and_avoid_dots()
tietools.tie_chain_to_augmented_tuplet_with_proportions_and_encourage_dots()
tietools.tie_chain_to_diminished_tuplet_with_proportions_and_avoid_dots()
tietools.tie_chain_to_diminished_tuplet_with_proportions_and_encourage_dots()
```

```
tietools.tie_chain_to_tuplet_with_ratio()
```

- Renamed:

```
tietools.iterate_nontrivial_tie_chains_forward_in_expr()
tietools.iterate_nontrivial_tie_chains_backward_in_expr()
tietools.iterate_pitched_tie_chains_forward_in_expr()
tietools.iterate_pitched_tie_chains_backward_in_expr()
tietools.iterate_tie_chains_forward_in_expr()
tietools.iterate_tie_chains_backward_in_expr()
```

```
tietools.iterate_nontrivial_tie_chains_in_expr(reverse=[True, False])
tietools.iterate_pitched_tie_chains_in_expr(reverse=[True, False])
tietools.iterate_tie_chains_in_expr(reverse=[True, False])
```

Changes to the `tuplettools` package:

- Renamed:

```
tuplettools.is_proper_tuplet_multiplier()
```

```
durationtools.is_proper_tuplet_multiplier()
```

- Renamed:

```
tuplettools.make_augmented_tuplet_from_duration_and_proportions_and_avoid_dots()
tuplettools.make_diminished_tuplet_from_duration_and_proportions_and_avoid_dots()
tuplettools.make_augmented_tuplet_from_duration_and_proportions_and_encourage_dots()
tuplettools.make_diminished_tuplet_from_duration_and_proportions_and_encourage_dots()
```

```
tuplettools.make_tuplet_from_durations_and_proportions(big_endian=[True, False])
```

Removed three packages.

- Removed `constrainttools` package.
- Removed `lyricstools` package.
- Removed `quantizationtools` package.

CHAPTER

SIXTYFOUR

BIBLIOGRAPHY

BIBLIOGRAPHY

[Adan2006] Víctor Adán. Music <-> Geometry <-> Meta-Music. Draft February 12, 2006.

[AgonAssayagBresson2006] Carlos Agon, Gérard Assayag, Jean Bresson. The OM Composer's Book 1. Éditions Delatour, Paris. 2006.

[AgonHaddadAssayag2002] Carlos Agon, Karim Haddad & Gerard Assayag. Répresentation et rendu de structures rhythmiques. Journées d'Informatique Musicale, 9th ed., Marseille, 29 - 31 May 2002.

[Alegant1993] Brian Alegant. The seventy-seven partitions of the aggregate: Analytical and theoretical implications. Doctoral Dissertation. The University of Rochester, Eastman School of Muisc. 1993.

[Ariza2005] Christopher Ariza. An Open Design for Computer-Aided Algorithmic Music Composition: athenaCL. Dissertation.com, Boca Raton. 2005.

[BacaAdan2007] Trevor Bača & Víctor Adán. Cuepatlahto and Lascaux: two approaches to the formalized control of musical score. Draft June 7, 2007.

[BressonAgonAssayag2008] Jean Bresson, Carlos Agon, Gérard Assayag. The OM Composer's Book 2. Éditions Delatour, Paris. 2008

[Carter2002] Eliot Carter. Harmony Book. Nicholas Hopkins and John F. Link, eds. Carl Fischer, New York. 2002.

[Haddad] Karim Haddad. Le Temps comme Territoire: pour une géographie temporelle.

[Kampela1998] Arthur Kampela. Uma Faca Só Lâmina. Doctoral Dissertation. Columbia University, NY, NY. 1998.

[Malt2008] Mikhaïl Malt. Some Considerations on Brian Ferneyhough's Musical Language Through His Use of CAC – Part I: Time and Rhythmic Structures. In Bresson, Agon and Assayag (2008).

[Morris1987] Robert Morris. Composition with Pitch-Classes. Yale University Press, New Haven. 1987.

[Nauert1997] Paul Nauert. Timespan Formation in Nonmetric, Posttonal Music. Doctoral Dissertation. Columbia University, NY, NY. 1997.

[NienhuysNieuwenhuizen2003] Han-Wen Nienhuys & Jan Nieuwenhuizen. Lilypond: A system for automated music engraving. Proceedings of the XIV Colloquium on Musical Informatics. Firenze, Italy. May 8 - 10, 2003.

[Ross1987] Ted Ross. Teach Yourself The Art of Music Engraving and Processing. Hansen House, Miami Beach. 1987.

[Selfridge-Field1997] Eleanor Selfridge-Field, ed. Beyond MIDI: The Handbook of Musical Codes. The MIT Press. Cambridge, Massachusetts. 1997.

[Valle] Andrea Valle. GeoGraphy: Notazione musicale e composizione algorithmica. Centro Interdipartimentale di Ricerca sulla Multimedialità e l'Audiovisivo. Università degli Studi di Torino.

[WulfsonBarrettWinter] Harris Wulfson, G. Douglas Barrett & Michael Winter. Automatic Notation Generators.