Abjad Documentation

Release 2.14

Trevor Bača, Josiah Wolf Oberholtzer, Víctor Adán

December 19, 2013

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 199,000 lines of code. About half of these implement the automated tests that check the correctness of Abjad. The rest of the code implements 58 packages comprising 459 classes and 526 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 2.7.5 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 2.17 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 contextualize environment variables.

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

There are different ways to install Python packages on your computer. One of the most direct ways is with `pip`, the package management tool recommended by the Python Package Index.

```
sudo pip install abjad --upgrade
```

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

If you don't have `pip`, but you do have Python's `easy_install` (as is often the case), we strongly recommend using `easy_install` to install `pip`, and then `pip` to install Abjad.

```
sudo easy_install pip
```

2.4 Manually installing Abjad from the Python Package Index

If you do not have `pip` or `easy_install` installed on your computer you then 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 Configuring Abjad

Abjad creates a `~/.abjad` directory the first time it runs. In `~/.abjad` you will find a the file `abjad.cfg`. This is the Abjad configuration file. You can use the Abjad configuration file to tell Abjad about your preferred PDF file viewer, MIDI player, your preferred LilyPond language and so on.

By default, your configuration file's contents will look approximately like this:

```
# Abjad configuration file created by Abjad on 19 October 2013 12:30:17.
# File is interpreted by ConfigObj and should follow ini syntax.

# Set to the directory where all Abjad-generated files
# (such as PDFs and LilyPond files) should be saved.
# Defaults to $HOME.abjad/output/
abjad_output = /Users/josiah/.abjad/output

# Default accidental spelling (mixed|sharps|flats).
accidental_spelling = mixed

# Comma-separated list of LilyPond files that
# Abjad will "\include" in all generated *.ly files
lilypond_includes = ,

# Language to use in all generated LilyPond files.
lilypond_language = english

# Lilypond executable path. Set to override dynamic lookup.
lilypond_path = lilypond

# MIDI player to open MIDI files.
# When unset your OS should know how to open MIDI files.
midi_player =

# PDF viewer to open PDF files.
# When unset your OS should know how to open PDFs.
pdf_viewer =

# Text editor to edit text files.
```

```
# When unset your OS should know how to open text files.
text_editor =
```

In Linux, for example, you might want to set your `pdf_viewer` to `evince` and your `midi_player` to `tiMIDIty`.

The configuration file is in `ini` syntax, so make sure to follow those conventions when editing.

Part II

System overview

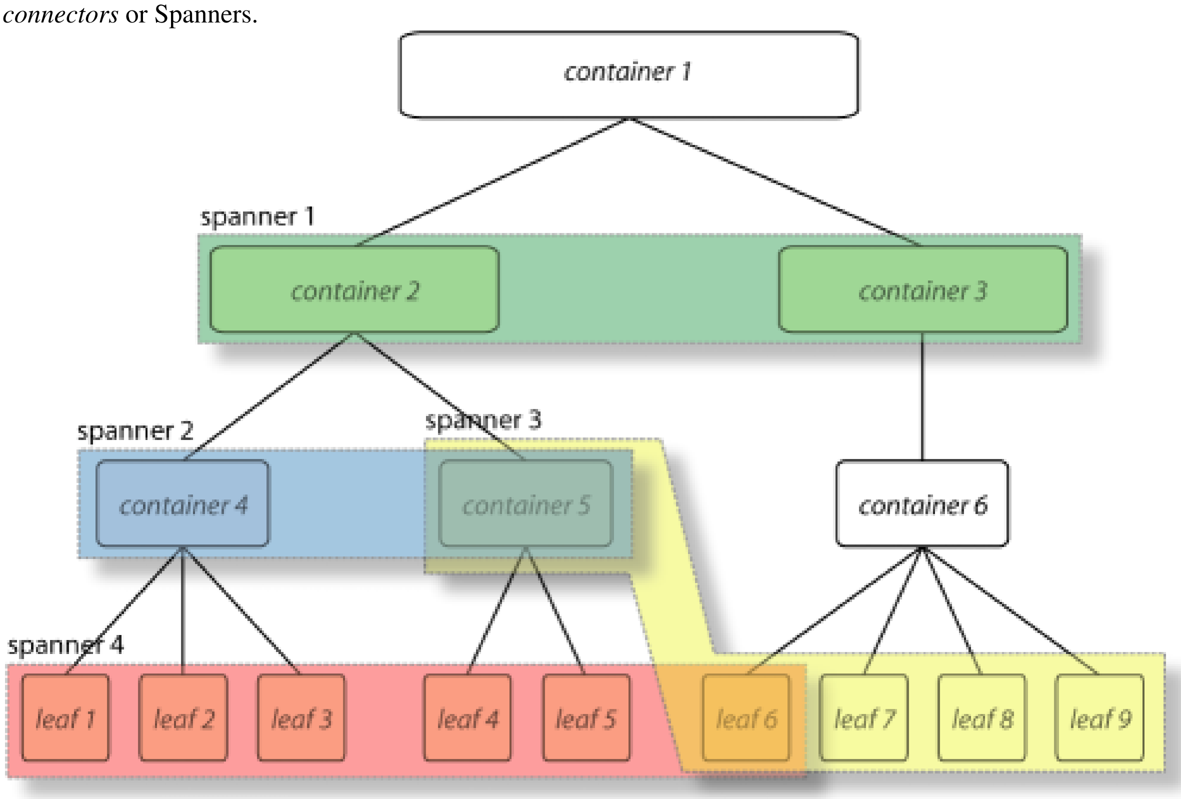
CHAPTER

THREE

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.

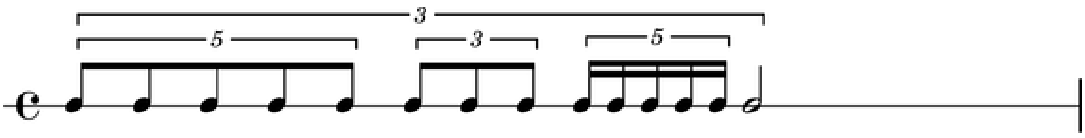
The LCSM model can be schematically visualized as a superposition of two complementary and completely independent layers of structure: a *tree* that includes the Containers and the Leaves, and a layer of free floating *connectors* or Spanners.



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.

3.1 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.



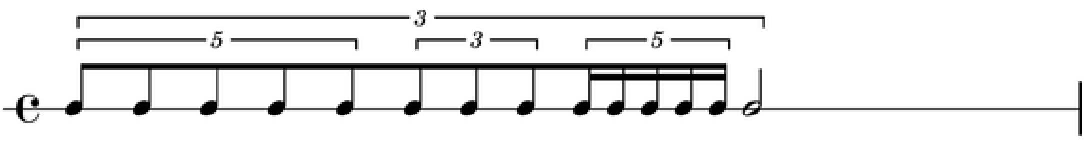
What we see in this little fragment is a measure with 4/4 meter, 14 notes and four tuplet brackets prolating the notes. The three bottom tuplets (with ratios 5:4, 3:2, 5:4) prolate all but the last note. The topmost tuplet prolates all the notes in the measure and combines with the bottom three tuplets 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.

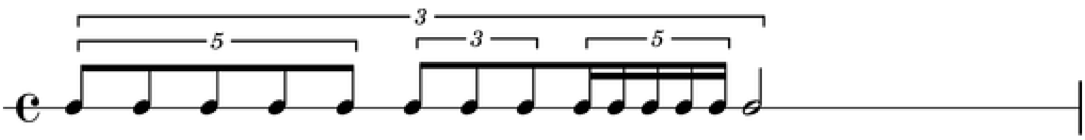
What we are describing here is a tree, and it is the structure of Abjad *containers*.

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:

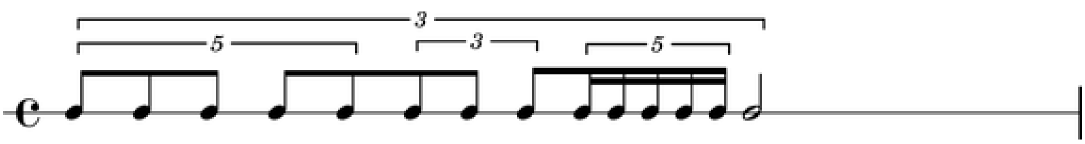
Beaming alternative 1:



Beaming alternative 2:



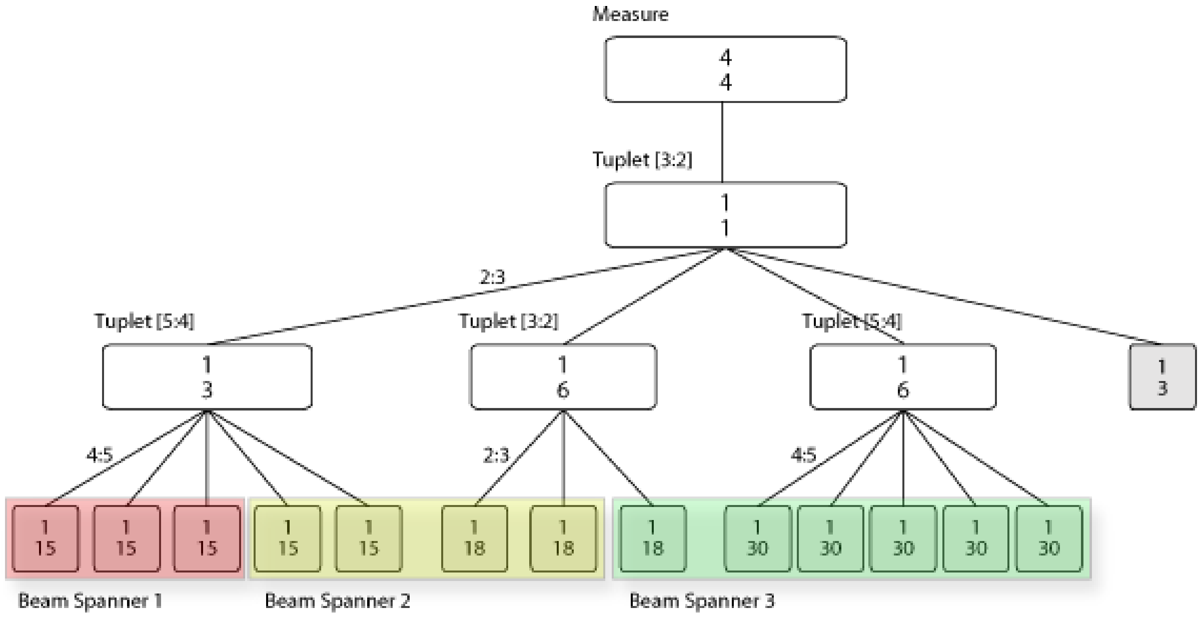
Beaming alternative 3:



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*.

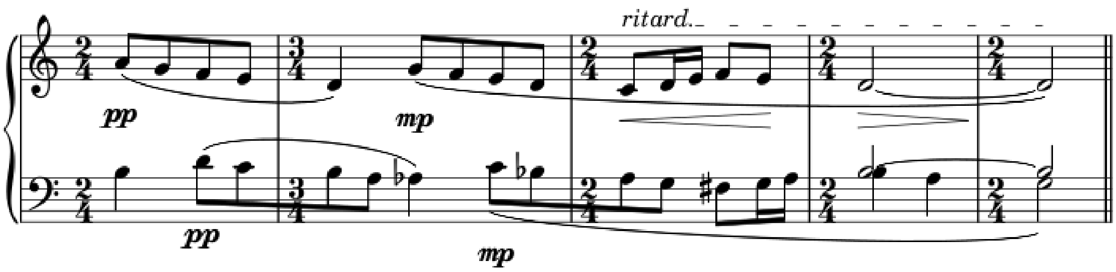
Below we have the score of our tuplet example with alternative beaming and its the Leaf-Container-Spanner graph. Notice that the colored blocks represent spanners.

Beaming alternative 3 (graph):



3.2 Example 2

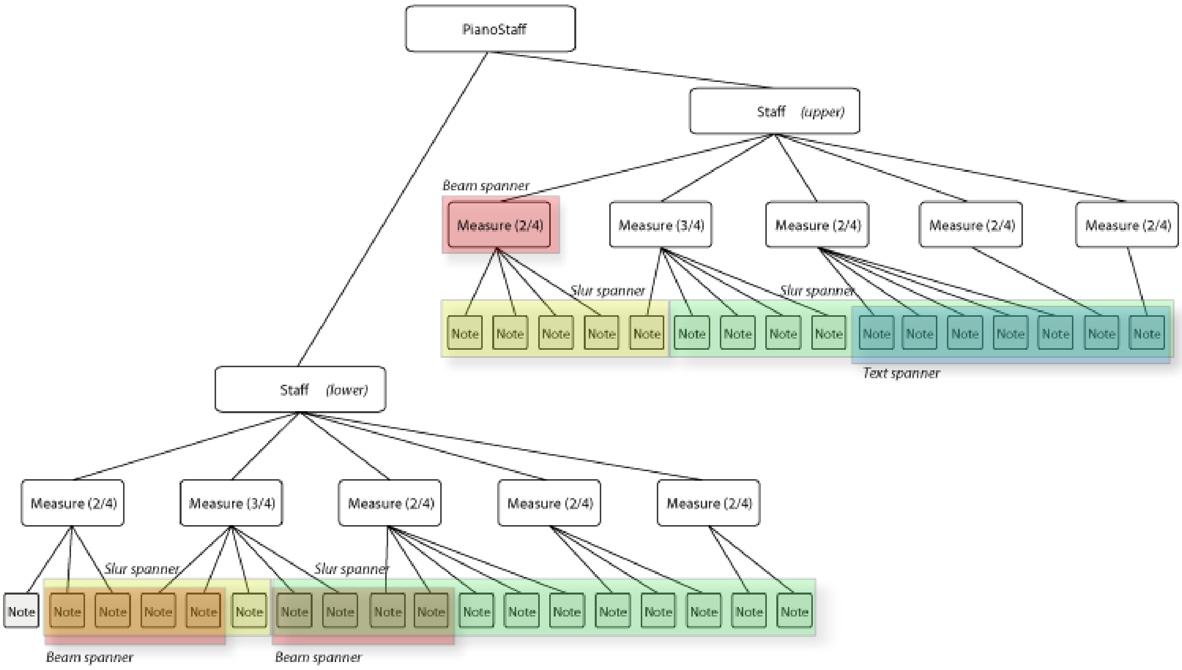
As a second example let's look at the last five measures of Bartók's *Wandering* from Mikrokosmos vol. III. As simple as it may seem, these five measures carry with them a lot of information pertaining to musical notation.



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, dynamicings, 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

FOUR

PARSING

Abjad provides a small number of domain-specific language parsers. The most important of these is its LilyPond parser.

4.1 LilyPond Parsing

Abjad's LilyPond parser parses a large (although incomplete) subset of LilyPond's syntax:

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

```
>>> string = r"""
... \new Score <<
...     \new StaffGroup <<
...         \new Staff {
...             r2 ^ \markup { \center-column { tutti \line { ( con sord. ) } } }
...             r8
...             es'' [ ( \ppp
...             fs'''
...             es'''
...             fs''' \flageolet
...             es'''
...             fs'''
...             es''
...             fs'' ] )
...             r
...             r4
...         }
...         \new Staff {
...             r4 ^ \markup { ( con sord. ) }
...             r8
...             es' [ ( \ppp
...             fs''
...             es'' ] )
...             r
...             es' [ (
...             fs''
...             es'
...             fs' ] )
...             r
...             fs'' [ (
...             es'
...             fs' ] )
...             r
...         }
...         \new Staff {
...             r8 ^ \markup { tutti }
...             ds' [ ( \ppp
...             es''
...             ds'' ]
...             es' [
...             ds'
...             es''
...             ds'' ] )
...             r4
```

```
...             es''8 [ (
...             ds'
...             es' ] )
...             r
...             es'' [ (
...             ds' ] )
...         }
...     >>
... >>
... """
```

```
>>> parsed = parser(string)
```

```
>>> f(parsed)
\new Score <<
    \new StaffGroup <<
        \new Staff {
            r2
                ^ \markup {
                    \center-column
                        {
                            tutti
                            \line
                                {
                                    (
                                    con
                                    sord.
                                    )
                                }
                        }
                    }
            r8
            es''8 \ppp [ (
            fs'''8
            es'''8
            fs'''8 -\flageolet
            es'''8
            fs'''8
            es''8
            fs''8 ] )
            r8
            r4
        }
        \new Staff {
            r4
                ^ \markup {
                    (
                    con
                    sord.
                    )
                    }
            r8
            es'8 \ppp [ (
            fs''8
            es''8 ] )
            r8
            es'8 [ (
            fs''8
            es'8
            fs'8 ] )
            r8
            fs''8 [ (
            es'8
            fs'8 ] )
            r8
        }
        \new Staff {
            r8 ^ \markup { tutti }
            ds'8 \ppp [ (
            es''8
            ds''8 ]
            es'8 [
```

```
            ds'8
            es''8
            ds''8 ] )
            r4
            es''8 [ (
            ds'8
            es'8 ] )
            r8
            es''8 [ (
            ds'8 ] )
        }
    >>
>>
```

```
>>> show(parsed)
```

The LilyPond parser understands most spanners, articulations and dynamics:

```
>>> 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 )
}
```

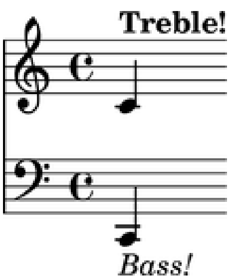
```
>>> show(result)
```

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 also understands certain aspects of LilyPond file layouts, such as 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)
% 2013-12-18 16:08

\version "2.17.97"
\language "english"

\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)
```

The LilyPond parser supports a small number of LilyPond music functions, such as `\relative` and `\transpose`.

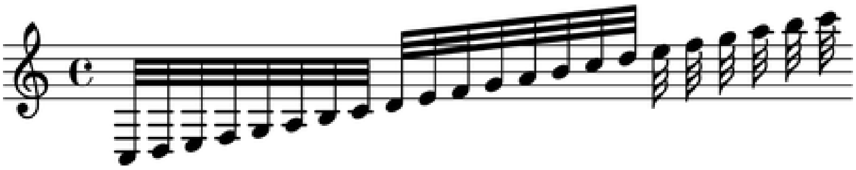
Music functions which mutate the score during compilation result in a normalized Abjad score structure. The resulting structure corresponds to the music as it appears on the page, rather than as it was input to the parser:

```
>>> 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)
```



4.2 RhythmTree Parsing

Abjad's rhythm-tree parser parses a microlanguage resembling Ircam's RTM Lisp syntax, and generates a sequence of RhythmTree structures, which can be furthered manipulated by composers, before being converted into an Abjad score object:

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

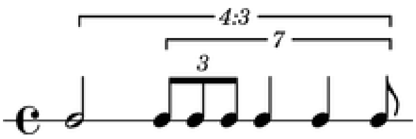
```
>>> string = '(3 (1 (1 ((2 (1 1 1)) 2 2 1))))'
>>> result = parser(string)
>>> result[0]
RhythmTreeContainer(
    children=(
        RhythmTreeLeaf(
            preprolated_duration=Duration(1, 1),
            is_pitched=True
            ),
        RhythmTreeContainer(
            children=(
                RhythmTreeContainer(
                    children=(
                        RhythmTreeLeaf(
                            preprolated_duration=Duration(1, 1),
                            is_pitched=True
                            ),
                        RhythmTreeLeaf(
                            preprolated_duration=Duration(1, 1),
                            is_pitched=True
                            ),
                        RhythmTreeLeaf(
                            preprolated_duration=Duration(1, 1),
                            is_pitched=True
                            ),
                        ),
                    preprolated_duration=Duration(2, 1)
                    ),
                RhythmTreeLeaf(
                    preprolated_duration=Duration(2, 1),
                    is_pitched=True
                    ),
                RhythmTreeLeaf(
                    preprolated_duration=Duration(2, 1),
                    is_pitched=True
                    ),
                RhythmTreeLeaf(
                    preprolated_duration=Duration(1, 1),
                    is_pitched=True
                    ),
                ),
```

```
            preprolated_duration=Duration(1, 1)
            ),
        ),
    preprolated_duration=Duration(3, 1)
    )
```

```
>>> tuplet = result[0]((1, 4))[0]
>>> f(tuplet)
\tweak #'text #tuplet-number::calc-fraction-text
\times 3/4 {
    c'2
    \times 4/7 {
        \times 2/3 {
            c'8
            c'8
            c'8
        }
        c'4
        c'4
        c'8
    }
}
```

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

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



4.3 "Reduced-Ly" Parsing

Abjad's "reduced-ly" parser 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. It also allows for rapidly notating notes and rests without needing to specify pitches. It is 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)
```

CHAPTER

FIVE

DURATIONS

5.1 Breves, longas and other long durations

A breve is a duration equal to two whole notes. Abjad supports breve-durated notes, rests and chords with and without dots.

You can create breves with a LilyPond input string:

```
>>> note_1 = Note(r"c'\breve")
>>> note_2 = Note(r"d'\breve.")
```

Or with an explicit duration:

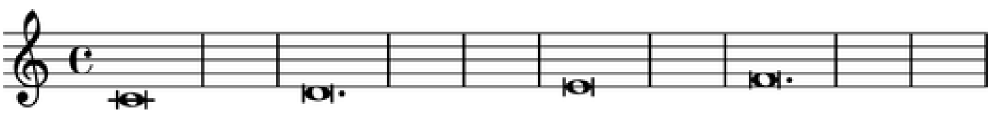
```
>>> note_3 = Note("e'", Duration(2, 1))
>>> note_4 = Note("f'", Duration(3, 1))
```

The written duration of a breve always returns an Abjad duration object:

```
>>> notes = [note_1, note_2, note_3, note_4]
>>> for note in notes:
...     note, note.written_duration
...
(Note("c'\\breve"), Duration(2, 1))
(Note("d'\\breve."), Duration(3, 1))
(Note("e'\\breve"), Duration(2, 1))
(Note("f'\\breve."), Duration(3, 1))
```

LilyPond renders breves like this:

```
>>> staff = Staff(notes)
>>> show(staff)
```

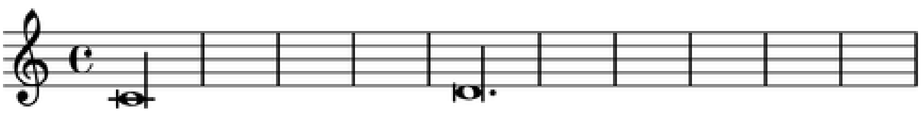


Abjad also supports longas. A longa equals two breves:

```
>>> note_1 = Note(r"c'\longa")
>>> note_2 = Note("d'", Duration(6, 1))
```

```
>>> notes = [note_1, note_2]
>>> for note in notes:
...     note, note.written_duration
...
(Note("c'\\longa"), Duration(4, 1))
(Note("d'\\longa."), Duration(6, 1))
```

```
>>> staff = Staff(notes)
>>> show(staff)
```



A maxima is a duration equal to two longas:

```
>>> note_1 = Note(r"c'\maxima")
>>> note_2 = Note("d''", Duration(12, 1))
```

```
>>> notes = [note_1, note_2]
>>> for note in notes:
...     note, note.written_duration
...
(Note("c'\\maxima"), Duration(8, 1))
(Note("d'\\maxima."), Duration(12, 1))
```

Abjad supports maximas and LilyPond supplies a `\maxima` command. But you can not use Abjad to render maxima-valued notes, rests and chords because LilyPond supplies no glyphs for these durations.

The same is true for all durations greater than or equal to eight whole notes: you can initialize and work with all such durations in Abjad but you will only be able to use LilyPond to render as notation those values equal to less than eight whole notes.

5.2 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 into calculations of duration.

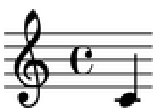
Abjad implements LilyPond multpliers as multiplier objects.

```
>>> note = Note("c'4")
>>> attach(Multiplier(1, 2), note)
```

```
>>> f(note)
c'4 * 1/2
```

```
>>> note.written_duration
Duration(1, 4)
>>> inspect(note).get_duration()
Duration(1, 8)
```

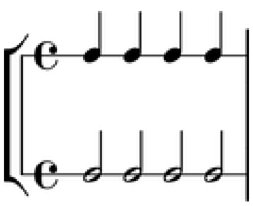
```
>>> show(note)
```



LilyPond multipliers scale the durations of the half notes below to that of quarter notes:

```
>>> quarter_notes = 4 * Note("c'4")
>>> half_note = Note("c'2")
>>> attach(Multiplier(1, 2), half_note)
>>> half_notes = 4 * half_note
>>> top_staff = scoretools.RhythmicStaff(quarter_notes)
>>> bottom_staff = scoretools.RhythmicStaff(half_notes)
>>> staff_group = scoretools.StaffGroup([top_staff, bottom_staff])
```

```
>>> show(staff_group)
```



Note that 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.

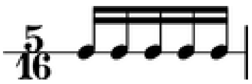
5.3 What's the difference between duration and written duration?

Abjad uses the term "written duration" to refer to the face value of notes, rests and chords prior to time-scaling effects of tuplets or measures with unusual time signatures. Abjad's written duration corresponds to the informal names most frequently used when talking about note duration.

Consider the measure below:

```
>>> measure = Measure((5, 16), "c16 c c c c")
>>> beam = Beam()
>>> attach(beam, [measure])
>>> staff = scoretools.RhythmicStaff([measure])
```

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



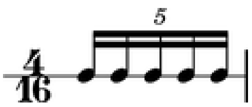
Every note in the measure equals one sixteenth of a whole note:

```
>>> note = measure[0]
>>> inspect(note).get_duration()
Duration(1, 16)
```

But now consider this measure:

```
>>> tuplet = Tuplet((4, 5), "c16 c c c c")
>>> measure = Measure((4, 16), [tuplet])
>>> beam = Beam()
>>> attach(beam, [measure])
>>> staff = scoretools.RhythmicStaff([measure])
```

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



The notes in this measure are equal to only one twentieth of a whole note: Every note in this measures

```
>>> note = tuplet[0]
>>> inspect(note).get_duration()
Duration(1, 20)
```

The notes in this measure are "sixteenth notes" with a duration equal to a value other than `1/16`. Abjad formalizes this distinction in the difference between the duration of these notes (`1/20`) and written duration of these notes (`1/16`).

Written duration is a user-assignable value. Users can assign and reassign the written duration of notes, rests and chords at initialization or any time thereafter. But the (unqualified) duration of a note, rest or chord is a derived property Abjad calculates based on the rhythmic context governing the note, rest or chord.

5.4 What does it mean for a duration to be "assignable"?

Western notation makes it easy to notate notes, rests and chords with durations like `1/4` and `3/16`. But notating notes, rests and chords with durations like `1/3` can only be done with recourse to tuplets or ties.

Abjad formalizes the difference between durations like `1/4` and `1/5` in the concept of **assignability**: a duration `n/d` is 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`.

Assignability is important because it explains why you can set the duration of any note, rest or chord to `1/4` but never to `1/5`.

Part III

Examples

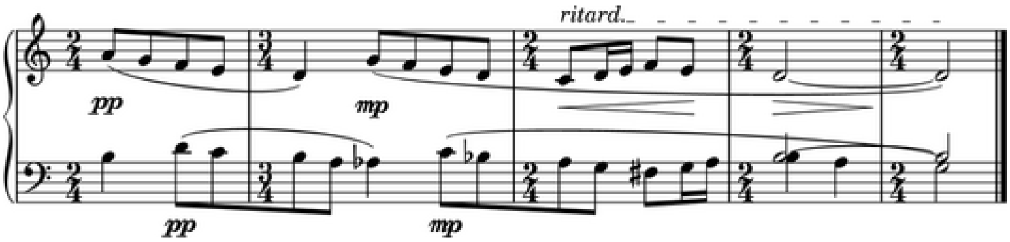
CHAPTER

SIX

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:



6.1 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)
```

6.2 The measures

Now let's add some empty measures:

```
>>> upper_measures = []
>>> upper_measures.append(Measure((2, 4), []))
>>> upper_measures.append(Measure((3, 4), []))
>>> upper_measures.append(Measure((2, 4), []))
>>> upper_measures.append(Measure((2, 4), []))
>>> upper_measures.append(Measure((2, 4), []))
```

```
>>> import copy
>>> lower_measures = copy.deepcopy(upper_measures)
```

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

6.3 The notes

Now let's add some notes.

We begin with the upper staff:

```
>>> upper_measures[0].extend("a'8 g'8 f'8 e'8")
>>> upper_measures[1].extend("d'4 g'8 f'8 e'8 d'8")
>>> upper_measures[2].extend("c'8 d'16 e'16 f'8 e'8")
>>> upper_measures[3].append("d'2")
>>> upper_measures[4].append("d'2")
```

The first three measures of the lower staff contain only one voice:

```
>>> lower_measures[0].extend("b4 d'8 c'8")
>>> lower_measures[1].extend("b8 a8 af4 c'8 bf8")
>>> lower_measures[2].extend("a8 g8 fs8 g16 a16")
```

The last two measures of the lower staff contain two voices each.

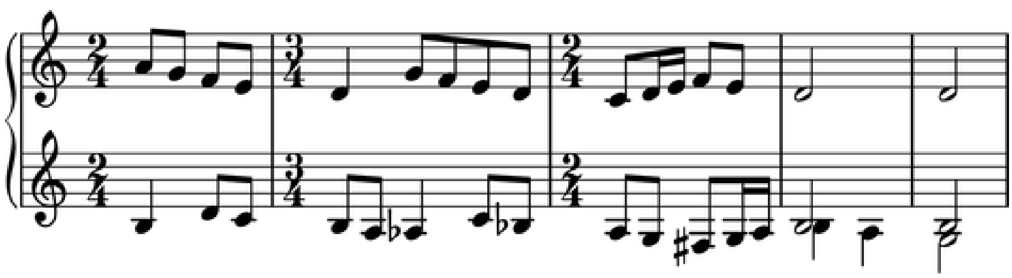
We use LilyPond `\voiceOne` and `\voiceTwo` commands to set the direction of stems in different voices. And we set `is_simltaneous` to true for each of the last two measures:

```
>>> upper_voice = Voice("b2", name='upper voice')
>>> command = indicatortools.LilyPondCommand('voiceOne')
>>> attach(command, upper_voice)
>>> lower_voice = Voice("b4 a4", name='lower voice')
>>> command = indicatortools.LilyPondCommand('voiceTwo')
>>> attach(command, lower_voice)
>>> lower_measures[3].extend([upper_voice, lower_voice])
>>> lower_measures[3].is_simultaneous = True
```

```
>>> upper_voice = Voice("b2", name='upper voice')
>>> command = indicatortools.LilyPondCommand('voiceOne')
>>> attach(command, upper_voice)
>>> lower_voice = Voice("g2", name='lower voice')
>>> command = indicatortools.LilyPondCommand('voiceTwo')
>>> attach(command, lower_voice)
>>> lower_measures[4].extend([upper_voice, lower_voice])
>>> lower_measures[4].is_simultaneous = True
```

Here's our work so far:

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



6.4 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:

```
>>> clef = Clef('bass')
>>> attach(clef, lower_staff)
```

Now let's add dynamics. 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 dynamicings to the second note of the first measure and the fourth note of the second measure:

```
>>> dynamic = Dynamic('pp')
>>> attach(dynamic, upper_measures[0][0])
```

```
>>> dynamic = Dynamic('mp')
>>> attach(dynamic, upper_measures[1][1])
```

```
>>> dynamic = Dynamic('pp')
>>> attach(dynamic, lower_measures[0][1])
```

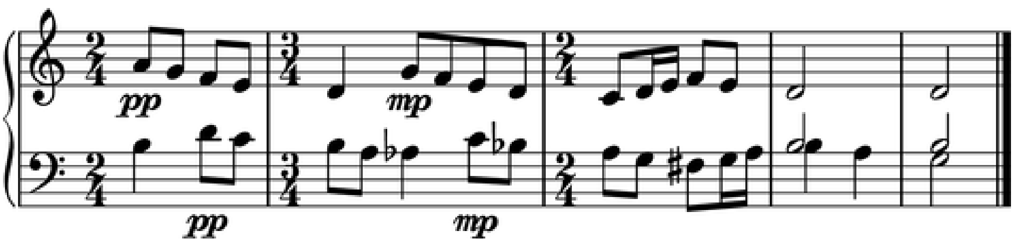
```
>>> dynamic = Dynamic('mp')
>>> attach(dynamic, lower_measures[1][3])
```

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

```
>>> score.add_double_bar()
BarLine('|.')
```

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:

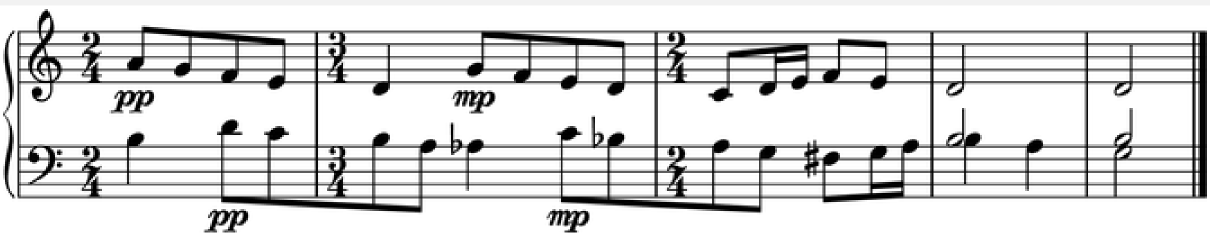
```
>>> upper_leaves = upper_staff.select_leaves(allow_discontiguous_leaves=True)
>>> lower_leaves = lower_staff.select_leaves(allow_discontiguous_leaves=True)
```

```
>>> beam = Beam()
>>> attach(beam, upper_leaves[:4])
```

```
>>> beam = Beam()
>>> attach(beam, lower_leaves[1:5])
```

```
>>> beam = Beam()
>>> attach(beam, lower_leaves[6:10])
```

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



Now some slurs:

```
>>> slur = Slur()
>>> attach(slur, upper_leaves[:5])
```

```
>>> slur = Slur()
>>> attach(slur, upper_leaves[5:])
```

```
>>> slur = Slur()
>>> attach(slur, lower_leaves[1:6])
```

Hairpins:

```
>>> crescendo = Crescendo()
>>> attach(crescendo, upper_leaves[-7:-2])
```

```
>>> decrescendo = Decrescendo()
>>> attach(decrescendo, upper_leaves[-2:])
```

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

```
>>> markup = Markup('ritard.')
>>> text_spanner = spannertools.TextSpanner()
>>> override(text_spanner).text_spanner.bound_details__left__text = markup
>>> attach(text_spanner, upper_leaves[-7:])
```

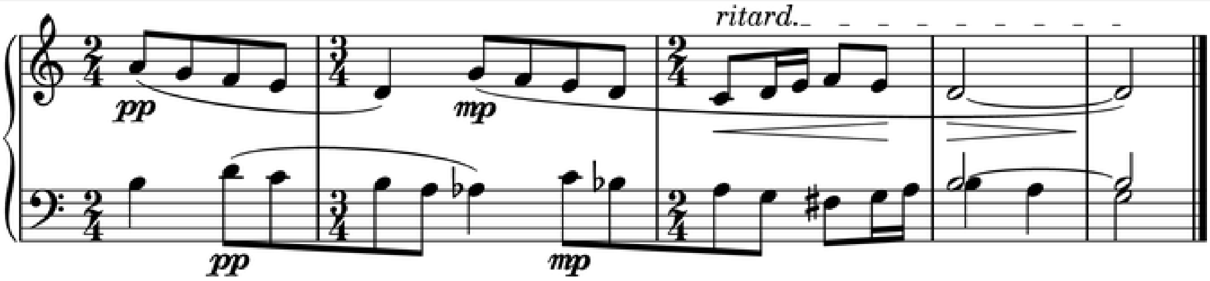
And ties connecting the last two notes in each staff:

```
>>> tie = Tie()
>>> attach(tie, upper_leaves[-2:])
```

```
>>> note_1 = lower_staff[-2]['upper voice'][0]
>>> note_2 = lower_staff[-1]['upper voice'][0]
>>> notes = [note_1, note_2]
>>> tie = Tie()
>>> attach(tie, notes)
```

The final result:

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



CHAPTER

SEVEN

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.

7.1 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)]
```

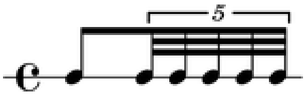
7.2 The transforms

Next we'll show how to divide a quarter note into various ratios, and then divide the final *logical tie* of the resulting tuplet into yet another ratio:

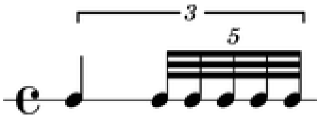
```
def make_nested_tuplet(
    tuplet_duration,
    outer_tuplet_proportions,
    inner_tuplet_subdivision_count,
    ):
    r'''Makes nested tuplet.
    '''

    outer_tuplet = Tuplet.from_duration_and_ratio(
        tuplet_duration, outer_tuplet_proportions)
    inner_tuplet_proportions = inner_tuplet_subdivision_count * [1]
    last_leaf = outer_tuplet.select_leaves()[-1]
    right_logical_tie = inspect(last_leaf).get_logical_tie()
    right_logical_tie.to_tuplet(inner_tuplet_proportions)
    return outer_tuplet
```

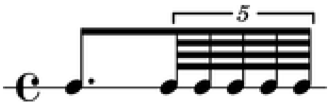
```
>>> tuplet = make_nested_tuplet(Duration(1, 4), (1, 1), 5)
>>> staff = scoretools.RhythmicStaff([tuplet])
>>> show(staff)
```




```
>>> tuplet = make_nested_tuplet(Duration(1, 4), (2, 1), 5)
>>> staff = scoretools.RhythmicStaff([tuplet])
>>> show(staff)
```



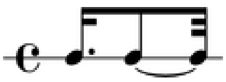

```
>>> tuplet = make_nested_tuplet(Duration(1, 4), (3, 1), 5)
>>> staff = scoretools.RhythmicStaff([tuplet])
>>> show(staff)
```



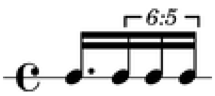
A *logical tie* is a selection of notes or chords connected by ties. It lets us talk about a notated rhythm of `5/16`, for example, which can not be expressed with only a single leaf.

Note how we can divide a tuplet whose outer proportions are `3/5`, where the second *logical tie* requires two notes to express the `5/16` duration:

```
>>> normal_tuplet = Tuplet.from_duration_and_ratio(Duration(1, 4), (3, 5))
>>> staff = scoretools.RhythmicStaff([normal_tuplet])
>>> show(staff)
```



```
>>> subdivided_tuplet = make_nested_tuplet(Duration(1, 4), (3, 5), 3)
>>> staff = scoretools.RhythmicStaff([subdivided_tuplet])
>>> show(staff)
```



7.3 The rhythms

Now that we know how to make the basic building block, let's make a lot of tuplets all at once.

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

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

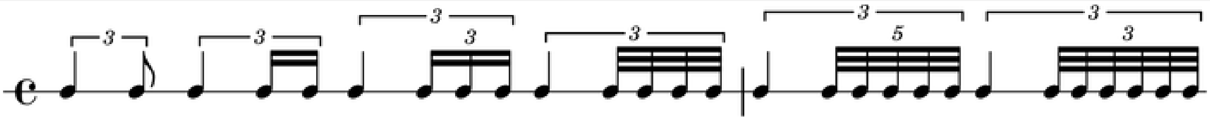
And then we make one row of rhythms, with the last *logical tie* increasingly subdivided:

```
def make_row_of_nested_tuplets(
    tuplet_duration,
    outer_tuplet_proportions,
    column_count,
    ):
    r'''Makes row of nested tuplets.
    '''

    assert 0 < column_count
    row_of_nested_tuplets = []
```

```
    for n in range(column_count):
        inner_tuplet_subdivision_count = n + 1
        nested_tuplet = make_nested_tuplet(
            tuplet_duration,
            outer_tuplet_proportions,
            inner_tuplet_subdivision_count,
            )
        row_of_nested_tuplets.append(nested_tuplet)
    return row_of_nested_tuplets
```

```
>>> tuplets = make_row_of_nested_tuplets(duration, (2, 1), 6)
>>> staff = scoretools.RhythmicStaff(tuplets)
>>> show(staff)
```



If we can make one single row of rhythms, we can make many rows of rhythms. Let's try:

```
def make_rows_of_nested_tuplets(tuplet_duration, row_count, column_count):
    r'''Makes rows of nested tuplets.
    '''

    assert 0 < row_count
    rows_of_nested_tuplets = []
    for n in range(row_count):
        outer_tuplet_proportions = (1, n + 1)
        row_of_nested_tuplets = make_row_of_nested_tuplets(
            tuplet_duration, outer_tuplet_proportions, column_count)
        rows_of_nested_tuplets.append(row_of_nested_tuplets)
    return rows_of_nested_tuplets
```

```
>>> score = Score()
>>> for tuplet_row in make_rows_of_nested_tuplets(duration, 4, 6):
...     score.append(scoretools.RhythmicStaff(tuplet_row))
...
>>> show(score)
```

That's getting close to what we want, but the typography isn't as good as it could be.

7.4 The score

First we'll package up the logic for making the un-styled score into a single function:

```
def make_score(tuplet_duration, row_count, column_count):
    r'''Makes score.
    '''

    score = Score()
    rows_of_nested_tuplets = make_rows_of_nested_tuplets(
```

```
        tuplet_duration, row_count, column_count)
    for row_of_nested_tuplets in rows_of_nested_tuplets:
        staff = scoretools.RhythmicStaff(row_of_nested_tuplets)
        time_signature = indicatortools.TimeSignature((1, 4))
        attach(time_signature, staff)
        score.append(staff)
    return score
```

```
>>> score = make_score(Duration(1, 4), 4, 6)
>>> show(score)
```

Then we'll apply some formatting overrides to improve its overall appearance:

```
def configure_score(score):
    r'''Configured score.
    '''

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

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

The proportional spacing makes the score much easier to read, but now the notation is much too big. We'll clean that up next.

7.5 The LilyPond file

Let's adjust the overall size of our output, and put everything together:

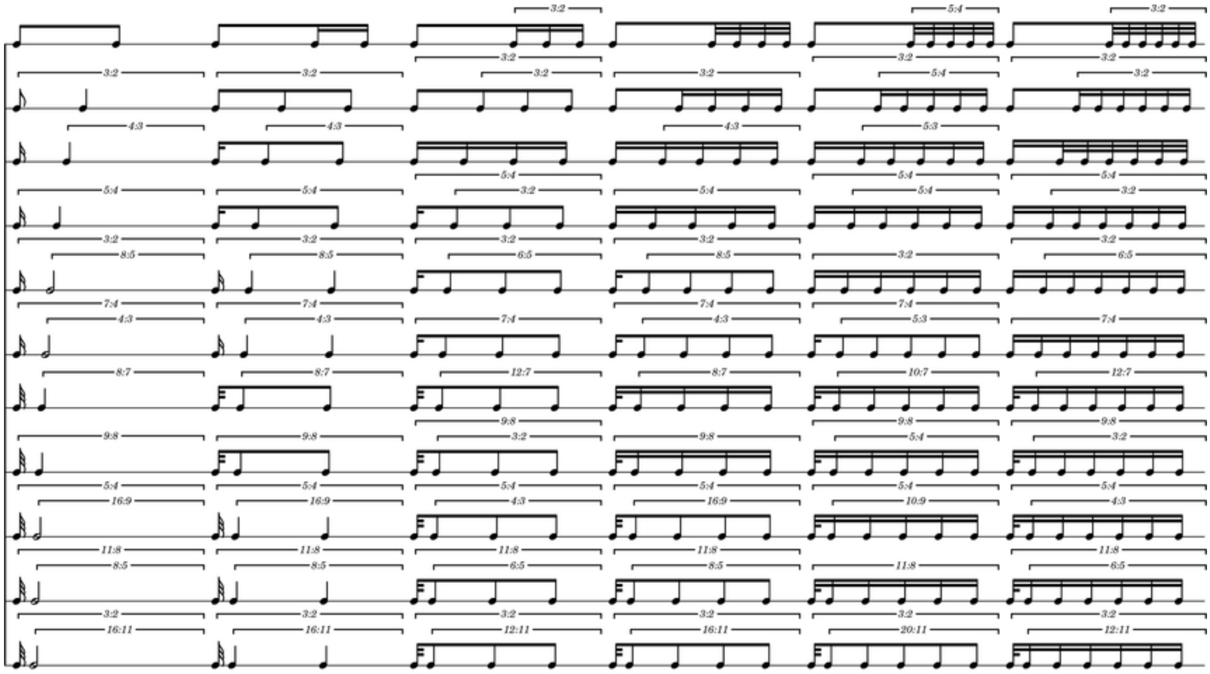
```
def make_lilypond_file(tuplet_duration, row_count, column_count):
    r'''Makes LilyPond file.
    '''

    score = make_score(tuplet_duration, row_count, column_count)
    configure_score(score)
    lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
    configure_lilypond_file(lilypond_file)
    return lilypond_file
```

```
def configure_lilypond_file(lilypond_file):
    r'''Configures LilyPond file.
    '''

    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
    spacing_vector = layouttools.make_spacing_vector(0, 0, 8, 0)
    lilypond_file.paper_block.system_system_spacing = spacing_vector
```

```
>>> lilypond_file = make_lilypond_file(Duration(1, 4), 11, 6)
>>> show(lilypond_file)
```



CHAPTER EIGHT

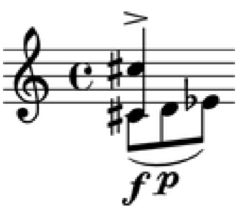
LIGETI: *DÉSORDRE*

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.

8.1 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 dynamicings 'f' 'p'.

The two "layers" of the *cell* we will model with two Voices inside a simultaneous 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 = scoretools.make_notes(pitches, [(1, 8)])
>>> beam = Beam()
>>> attach(beam, notes)
>>> slur = Slur()
>>> attach(slur, notes)
>>> dynamic = Dynamic('f')
>>> attach(dynamic, notes[0])
>>> dynamic = Dynamic('p')
>>> attach(dynamic, notes[1])
```

```
>>> voice_lower = Voice(notes)
>>> voice_lower.name = 'rh_lower'
>>> command = indicatortools.LilyPondCommand('voiceTwo')
>>> attach(command, voice_lower)
```

The notes belonging to the eighth note run are first beamed and slurred. Then we add the dynamics 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))
>>> articulation = Articulation('>')
>>> attach(articulation, chord)
```

```
>>> voice_higher = Voice([chord])
>>> voice_higher.name = 'rh_higher'
>>> command = indicatortools.LilyPondCommand('voiceOne')
>>> attach(command, voice_higher)
```

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 simultaneous container:

```
>>> container = Container([voice_lower, voice_higher])
>>> container.is_simultaneous = 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 = [scoretools.Note(pitch, (1, 8)) for pitch in pitches]
    beam = spannertools.Beam()
    attach(beam, notes)
    slur = spannertools.Slur()
    attach(slur, notes)
    clef = indicatortools.Dynamic('f')
    attach(clef, notes[0])
    dynamic = indicatortools.Dynamic('p')
    attach(dynamic, notes[1])

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

    # make the upper voice
    upper_voice = scoretools.Voice([chord])
    upper_voice.name = 'RH Upper Voice'
    command = indicatortools.LilyPondCommand('voiceOne')
    attach(command, upper_voice)

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

    # make all 1/8 beats breakable
    for leaf in lower_voice.select_leaves()[:-1]:
        bar_line = indicatortools.BarLine('')
        attach(bar_line, 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.

8.2 The measure

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

```
def make_desordre_measure(pitches):
    '''Makes a measure composed of *Désordre cells*.

    `pitches` is a list of lists of number (e.g., [[1, 2, 3], [2, 3, 4]])

    The function returns a measure.
    '''
```

```
    for sequence in pitches:
        container = make_desordre_cell(sequence)
        time_signature = inspect(container).get_duration()
        time_signature = mathtools.NonreducedFraction(time_signature)
        time_signature = time_signature.with_denominator(8)
        measure = scoretools.Measure(time_signature, [container])

    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)
```

8.3 The staff

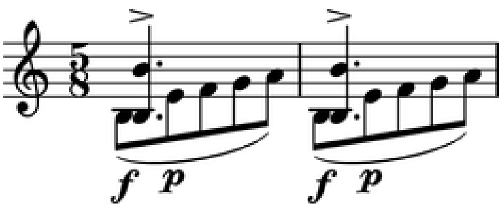
Now we move up to the next level, the staff:

```
def make_desordre_staff(pitches):
    r'''Makes Désordre staff.
    '''

    staff = scoretools.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)
```



8.4 The score

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

```
def make_desordre_score(pitches):
    '''Returns a complete piano staff 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
    clef = indicatortools.Clef('bass')
    attach(clef, piano_staff[1])
    key_signature = KeySignature('b', 'major')
    attach(key_signature, piano_staff[1])

    # wrap the piano staff in a score
    score = scoretools.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, -1, 0, 2, 4, 5, 7]],
...     [[-3, 2, 4], [-3, 2, 4, 5, 7]],
...     [[2, 5, 7], [-3, 9, 11, 12, 14]],
...     [[4, 5, 7, 9, 11], [2, 4, 5]],
...     [[-5, 4, 5, 7, 9, 11, 12]],
...     [[2, 9, 11], [2, 9, 11, 12, 14]],
...     ]
```

```
>>> bottom = [
...     [[-9, -4, -2], [-9, -4, -2, 1, 3]],
...     [[-6, -2, 1], [-9, -4, -2, 1, 3]],
...     [[-4, -2, 1, 3, 6], [-4, -2, 1]],
...     [[-9, -6, -4, -2, 1, 3, 6, 1]],
...     [[-6, -2, 1], [-6, -2, 1, 3, -2]],
...     [[-4, 1, 3], [-6, 3, 6, -6, -4]],
...     [[-14, -11, -9, -6, -4], [-14, -11, -9]],
...     [[-11, -2, 1, -6, -4, -2, 1, 3]],
...     [[-6, 1, 3], [-6, -4, -2, 1, 3]],
...     ]
```

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

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

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


f
p

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 contextualize 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

NINE

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.

9.1 The materials

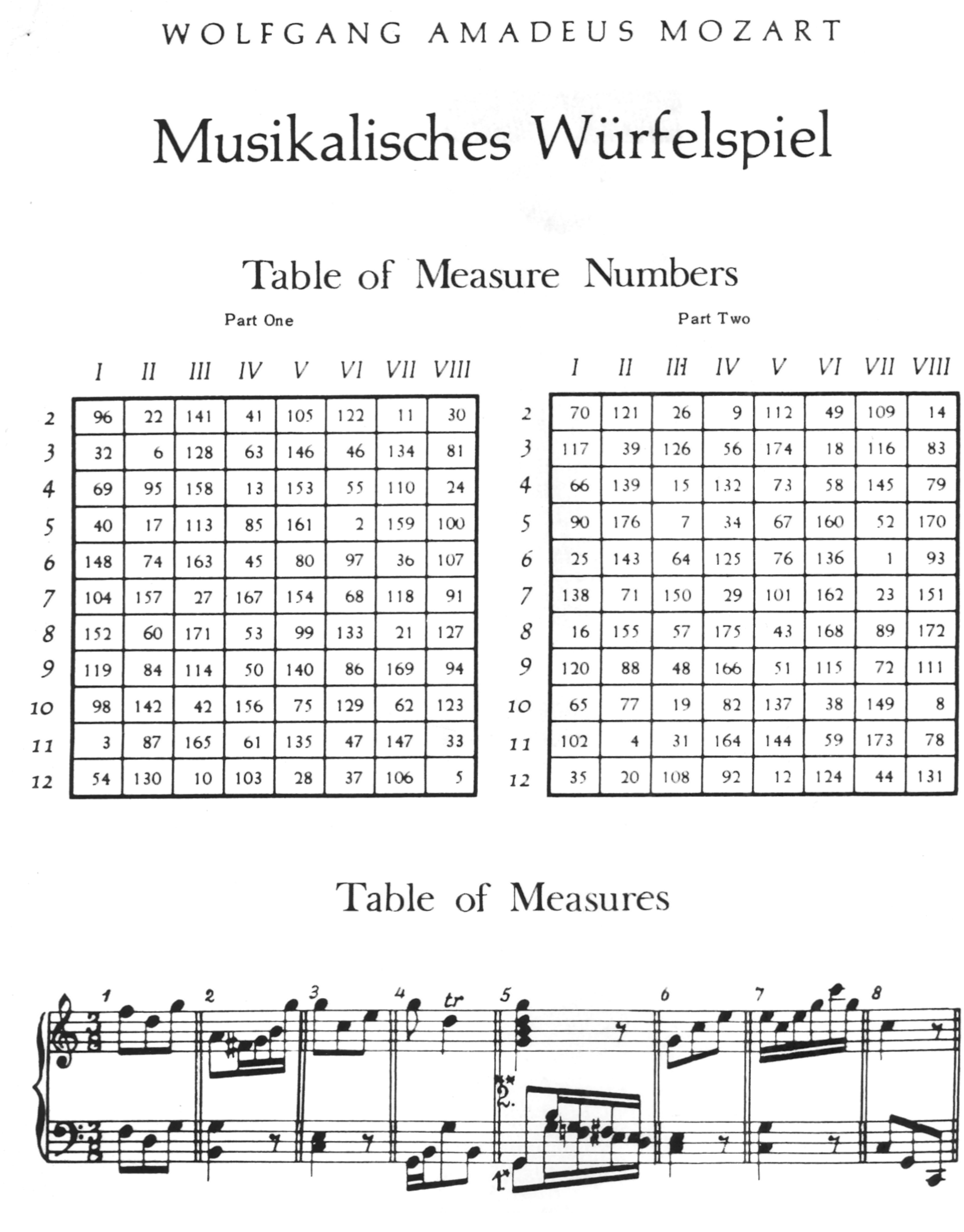
At the heart of the dice game is a large collection, *or corpus*, of musical fragments. Each fragment is a single 3/8 measure, consisting of a treble voice and a bass voice. Traditionally, these fragments are stored in a "score", or "table of measures", and located via two tables of measure numbers,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

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



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

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

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():
    r'''Makes 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):
    r'''Makes Mozart measure.
    '''

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

9.2 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():
    r'''Chooses 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"}
```

9.3 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 `LilyPondCommand` class, which allows you to place LilyPond commands like "break" relative to any score component:

```
>>> container = Container("c'4 d'4 e'4 f'4")
>>> command = indicatortools.LilyPondCommand('before-the-container', 'before')
>>> attach(command, container)
>>> command = indicatortools.LilyPondCommand('after-the-container', 'after')
>>> attach(command, container)
>>> command = indicatortools.LilyPondCommand('opening-of-the-container', 'opening')
>>> attach(command, container)
>>> command = indicatortools.LilyPondCommand('closing-of-the-container', 'closing')
>>> attach(command, container)
>>> command = indicatortools.LilyPondCommand('to-the-right-of-a-note', 'right')
>>> attach(command, container[2])
>>> f(container)
\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 `LilyPondCommand` 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 `LilyPondCommand` 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():
    r'''Makes Mozart score.
    '''

    score_template = templatetools.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)

    # attach indicators to the volta containers
    command = indicatortools.LilyPondCommand(
        'repeat volta 2', 'before'
        )
    attach(command, treble_volta)
    command = indicatortools.LilyPondCommand(
        'repeat volta 2', 'before'
        )
    attach(command, bass_volta)

    # append 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)

    # attach indicators to the alternative containers
    command = indicatortools.LilyPondCommand(
        'alternative', 'before'
        )
    attach(command, treble_alternative)
    command = indicatortools.LilyPondCommand(
        'alternative', 'before'
        )
    attach(command, bass_alternative)

    # append 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)

    # attach indicators
    time_signature = indicatortools.TimeSignature((3, 8))
    attach(time_signature, score['RH Staff'])
    bar_line = indicatortools.BarLine('|.')
    attach(bar_line, score['RH Voice'][-1])
    bar_line = indicatortools.BarLine('|.')
    attach(bar_line, score['LH Voice'][-1])

    # remove the old, default piano instrument attached to the piano staff
    # and attach a custom instrument mark
    detach(instrumenttools.Instrument, score['Piano Staff'])

    klavier = instrumenttools.Piano(
        instrument_name='Katzenklavier',
        short_instrument_name='kk.',
        )
    attach(klavier, 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...

9.4 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():
    r'''Makes 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)
```


Ein Musikalisches Wuerfelspiel
W. A. Mozart (maybe?)
Katzenklavier
Kk.

CHAPTER

TEN

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():
    r'''Makes Pärt 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_dynamics(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
```

10.1 The score template

```
class PartCantusScoreTemplate(abctools.AbjadObject):
    r'''Pärt Cantus score template.
    '''

    ### SPECIAL METHODS ###

    def __call__(self):
        '''Calls score template.

        Returns LilyPond file.
        '''

        # make bell voice and staff
        bell_voice = scoretools.Voice(name='Bell Voice')
        bell_staff = scoretools.Staff([bell_voice], name='Bell Staff')
        clef = indicatortools.Clef('treble')
```

```
        attach(clef, bell_staff)
        bells = instrumenttools.Instrument(
            instrument_name='Campana in La',
            short_instrument_name='Camp.',
            pitch_range='[C4, C6]',
            )
        attach(bells, bell_staff)
        tempo = indicatortools.Tempo((1, 4), (112, 120))
        attach(tempo, bell_staff)
        time_signature = indicatortools.TimeSignature((6, 4))
        attach(time_signature, bell_staff)

        # make first violin voice and staff
        first_violin_voice = scoretools.Voice(name='First Violin Voice')
        first_violin_staff = scoretools.Staff(
            [first_violin_voice],
            name='First Violin Staff',
            )
        clef = indicatortools.Clef('treble')
        attach(clef, first_violin_staff)
        violin = instrumenttools.Violin(
            instrument_name_markup='Violin I',
            short_instrument_name_markup='Vl. I'
            )
        attach(violin, first_violin_staff)

        # make second violin voice and staff
        second_violin_voice = scoretools.Voice(name='Second Violin Voice')
        second_violin_staff = scoretools.Staff(
            [second_violin_voice],
            name='Second Violin Staff',
            )
        clef = indicatortools.Clef('treble')
        attach(clef, second_violin_staff)
        violin = instrumenttools.Violin(
            instrument_name_markup='Violin II',
            short_instrument_name_markup='Vl. II'
            )
        attach(violin, second_violin_staff)

        # make viola voice and staff
        viola_voice = scoretools.Voice(name='Viola Voice')
        viola_staff = scoretools.Staff([viola_voice], name='Viola Staff')
        clef = indicatortools.Clef('alto')
        attach(clef, viola_staff)
        viola = instrumenttools.Viola()
        attach(viola, viola_staff)

        # make cello voice and staff
        cello_voice = scoretools.Voice(name='Cello Voice')
        cello_staff = scoretools.Staff([cello_voice], name='Cello Staff')
        clef = indicatortools.Clef('bass')
        attach(clef, cello_staff)
        cello = instrumenttools.Cello()
        attach(cello, cello_staff)

        # make bass voice and staff
        bass_voice = scoretools.Voice(name='Bass Voice')
        bass_staff = scoretools.Staff([bass_voice], name='Bass Staff')
        clef = indicatortools.Clef('bass')
        attach(clef, bass_staff)
        contrabass = instrumenttools.Contrabass(
            short_instrument_name_markup='Cb.'
            )
        attach(contrabass, 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
```

10.2 The bell music

```
def add_bell_music_to_score(score):
    r'''Adds bell music to score.
    '''

    bell_voice = score['Bell Voice']

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

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

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

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

10.3 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):
    r'''Adds string music to 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
    strings_staff_group = score['Strings Staff Group']
    for voice in  iterate(strings_staff_group).by_class(scoretools.Voice):
        shards = mutate(voice[:]).split([(6, 4)], cyclic=True)
        for shard in shards:
            scoretools.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():
    r'''Creates pitch contour reservoir.
    '''

    scale = tonalanalysistools.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_pitch_set_in_pitch_range(pitch_range)
        pitches = sorted(pitch_set, 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):
    r'''Shadows pitch contour reservoir.
    '''
```

```
    shadow_pitch_lookup = {
        pitchtools.NamedPitchClass('a'): -5, # add a P4 below
        pitchtools.NamedPitchClass('g'): -3, # add a m3 below
        pitchtools.NamedPitchClass('f'): -1, # add a m2 below
        pitchtools.NamedPitchClass('e'): -4, # add a M3 below
        pitchtools.NamedPitchClass('d'): -2, # add a M2 below
        pitchtools.NamedPitchClass('c'): -3, # add a m3 below
        pitchtools.NamedPitchClass('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_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_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):
    r'''Durates 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 = scoretools.MultimeasureRest((3, 2)) * div
        if mod:
            initial_rest += scoretools.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(scoretools.make_leaves([pitch], [durations[counter]]))
                counter = (counter + 1) % 2
            durated_contours.append(tuple(contour))

        durated_reservoir[instrument_name] = tuple(durated_contours)

    return durated_reservoir
```

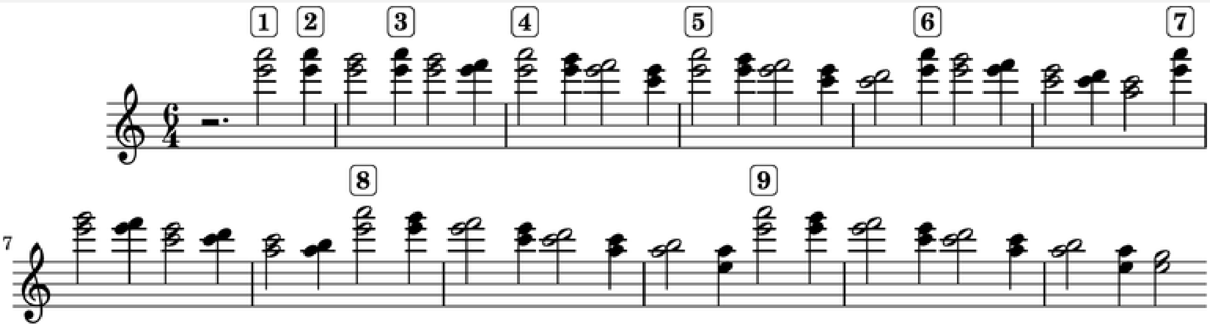
Let's see what a few of those look like. First, we'll build the entire reservoir from scratch, so you can see the process:

```
>>> 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,
...         )
...     attach(markup, descent[0])
...
```

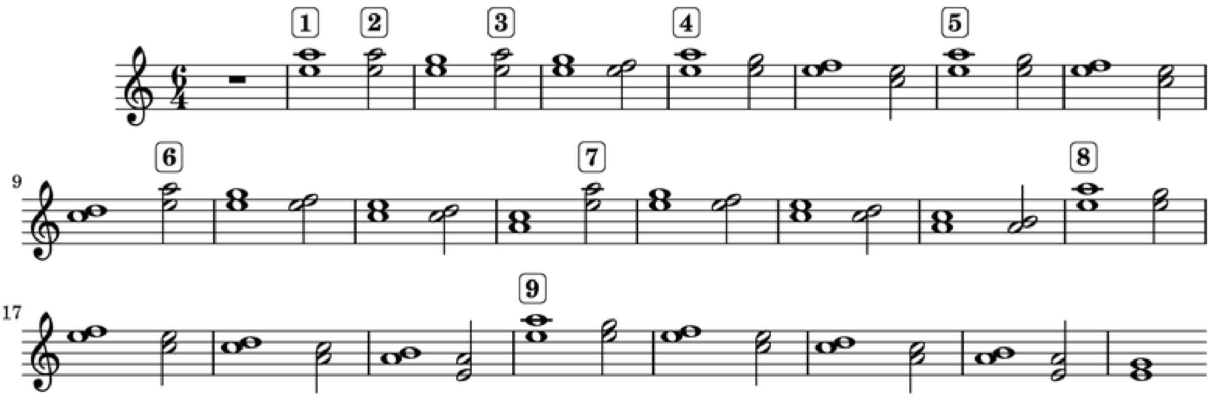
```
>>> staff = Staff(sequencetools.flatten_sequence(descents))
>>> time_signature = TimeSignature((6, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```



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,
...         )
...     attach(markup, descent[0])
...
```

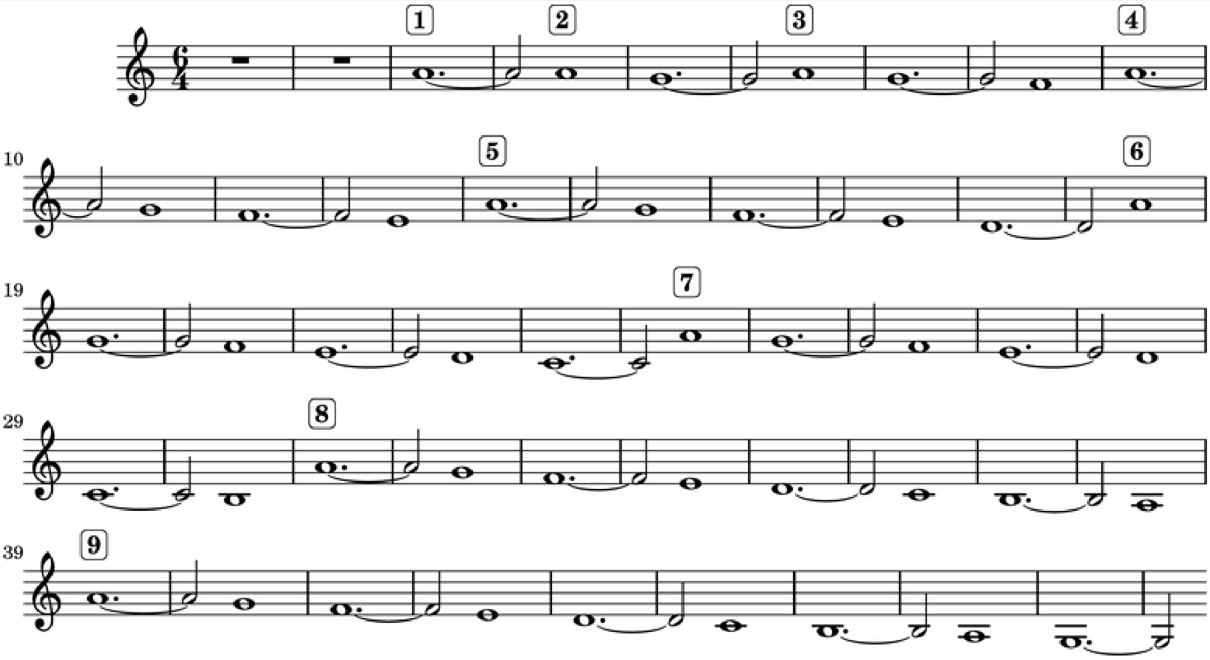
```
>>> staff = Staff(sequencetools.flatten_sequence(descents))
>>> time_signature = TimeSignature((6, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```



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,
...         )
...     attach(markup, descent[0])
...
```

```
>>> staff = Staff(sequencetools.flatten_sequence(descents))
>>> shards = mutate(staff[:]).split([(3, 2)], cyclic=True)
>>> time_signature = indicatortools.TimeSignature((6, 4))
>>> attach(time_signature, staff)
>>> show(staff)
```



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

10.4 The edits

```
def edit_first_violin_voice(score, durated_reservoir):
    r'''Edits first violin voice.
    '''

    voice = score['First Violin Voice']
```

```
    descents = durated_reservoir['First Violin']
    descents = selectiontools.ContiguousSelection(descents)

    last_descent = select(descents[-1], contiguous=True)
    copied_descent = mutate(last_descent).copy()
    voice.extend(copied_descent)

    final_sustain_rhythm = [(6, 4)] * 43 + [(1, 2)]
    final_sustain_notes = scoretools.make_notes(["c'"], final_sustain_rhythm)
    voice.extend(final_sustain_notes)
    tie = spannertools.Tie()
    attach(tie, final_sustain_notes)
    voice.extend('r4 r2.')
```

```
def edit_second_violin_voice(score, durated_reservoir):
    r'''Edits second violin voice.
    '''

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

    last_descent = select(descents[-1], contiguous=True)
    copied_descent = mutate(last_descent).copy()
    copied_descent = list(copied_descent)
    copied_descent[-1].written_duration = durationtools.Duration(1, 1)
    copied_descent.append(scoretools.Note('a2'))
    for leaf in copied_descent:
        articulation = indicatortools.Articulation('accent')
        attach(articulation, leaf)
        articulation = indicatortools.Articulation('tenuto')
        attach(articulation, leaf)
    voice.extend(copied_descent)

    final_sustain = []
    for _ in range(32):
        final_sustain.append(scoretools.Note('a1.'))
    final_sustain.append(scoretools.Note('a2'))
    articulation = indicatortools.Articulation('accent')
    attach(articulation, final_sustain[0])
    articulation = indicatortools.Articulation('tenuto')
    attach(articulation, final_sustain[0])

    voice.extend(final_sustain)
    tie = spannertools.Tie()
    attach(tie, final_sustain)
    voice.extend('r4 r2.')
```

```
def edit_viola_voice(score, durated_reservoir):
    r'''Edits viola voice.
    '''

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

    for leaf in descents[-1]:
        articulation = indicatortools.Articulation('accent')
        attach(articulation, leaf)
        articulation = indicatortools.Articulation('tenuto')
        attach(articulation, leaf)
    last_descent = select(descents[-1], contiguous=True)
    copied_descent = mutate(last_descent).copy()
    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 = scoretools.Note('e1')
    articulation = indicatortools.Articulation('tenuto')
    attach(articulation, bridge)
    articulation = indicatortools.Articulation('accent')
```

```
    attach(articulation, bridge)
    voice.append(bridge)

    final_sustain_rhythm = [(6, 4)] * 21 + [(1, 2)]
    final_sustain_notes = scoretools.make_notes(['e'], final_sustain_rhythm)
    articulation = indicatortools.Articulation('accent')
    attach(articulation, final_sustain_notes[0])
    articulation = indicatortools.Articulation('tenuto')
    attach(articulation, final_sustain_notes[0])
    voice.extend(final_sustain_notes)
    tie = spannertools.Tie()
    attach(tie, final_sustain_notes)
    voice.extend('r4 r2.')
```

```
def edit_cello_voice(score, durated_reservoir):
    r'''Edits cello voice.
    '''

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

    logical_tie = inspect(voice[-1]).get_logical_tie()
    for leaf in logical_tie.leaves:
        parent = leaf._get_parentage().parent
        index = parent.index(leaf)
        parent[index] = scoretools.Chord(['e,', 'a,'], leaf.written_duration)

    selection = voice[-len(descents[-1]):]
    unison_descent = mutate(selection).copy()
    voice.extend(unison_descent)
    for chord in unison_descent:
        index = inspect(chord).get_parentage().parent.index(chord)
        parent[index] = scoretools.Note(
            chord.written_pitches[1], chord.written_duration)
        articulation = indicatortools.Articulation('accent')
        attach(articulation, parent[index])
        articulation = indicatortools.Articulation('tenuto')
        attach(articulation, parent[index])

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

```
def edit_bass_voice(score, durated_reservoir):
    r'''Edits bass voice.
    '''

    voice = score['Bass Voice']

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

10.5 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):
    r'''Applies bowing marks to 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(iterate(measure).by_class(Chord)):
            if i % 2 == 0:
                articulation = indicatortools.Articulation('downbow')
                attach(articulation, chord)
            else:
                articulation = indicatortools.Articulation('upbow')
                attach(articulation, 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'),
            ]))
    markup = copy.copy(rebow_markup)
    attach(markup, score['First Violin Voice'][64][0])
    markup = copy.copy(rebow_markup)
    attach(markup, score['Second Violin Voice'][75][0])
    markup = copy.copy(rebow_markup)
    attach(markup, score['Viola Voice'][86][0])
```

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

```
def apply_dynamics(score):
    r'''Applies dynamics to score.
    '''

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

    voice = score['First Violin Voice']
    dynamic = indicatortools.Dynamic('ppp')
    attach(dynamic, voice[6][1])
    dynamic = indicatortools.Dynamic('pp')
    attach(dynamic, voice[15][0])
    dynamic = indicatortools.Dynamic('p')
    attach(dynamic, voice[22][3])
    dynamic = indicatortools.Dynamic('mp')
    attach(dynamic, voice[31][0])
    dynamic = indicatortools.Dynamic('mf')
```

```
    attach(dynamic, voice[38][3])
    dynamic = indicatortools.Dynamic('f')
    attach(dynamic, voice[47][0])
    dynamic = indicatortools.Dynamic('ff')
    attach(dynamic, voice[55][2])
    dynamic = indicatortools.Dynamic('fff')
    attach(dynamic, voice[62][2])

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

    voice = score['Viola Voice']
    dynamic = indicatortools.Dynamic('p')
    attach(dynamic, voice[8][0])
    dynamic = indicatortools.Dynamic('mp')
    attach(dynamic, voice[19][1])
    dynamic = indicatortools.Dynamic('mf')
    attach(dynamic, voice[30][0])
    dynamic = indicatortools.Dynamic('f')
    attach(dynamic, voice[36][0])
    dynamic = indicatortools.Dynamic('f')
    attach(dynamic, voice[42][0])
    dynamic = indicatortools.Dynamic('ff')
    attach(dynamic, voice[52][0])
    dynamic = indicatortools.Dynamic('fff')
    attach(dynamic, voice[62][0])

    voice = score['Cello Voice']
    dynamic = indicatortools.Dynamic('p')
    attach(dynamic, voice[10][0])
    dynamic = indicatortools.Dynamic('mp')
    attach(dynamic, voice[21][0])
    dynamic = indicatortools.Dynamic('mf')
    attach(dynamic, voice[31][0])
    dynamic = indicatortools.Dynamic('f')
    attach(dynamic, voice[43][0])
    dynamic = indicatortools.Dynamic('ff')
    attach(dynamic, voice[52][1])
    dynamic = indicatortools.Dynamic('fff')
    attach(dynamic, voice[62][0])

    voice = score['Bass Voice']
    dynamic = indicatortools.Dynamic('mp')
    attach(dynamic, voice[14][0])
    dynamic = indicatortools.Dynamic('mf')
    attach(dynamic, voice[27][0])
    dynamic = indicatortools.Dynamic('f')
    attach(dynamic, voice[39][0])
    dynamic = indicatortools.Dynamic('ff')
    attach(dynamic, voice[51][0])
    dynamic = indicatortools.Dynamic('fff')
    attach(dynamic, voice[62][0])
```

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

```
def apply_expressive_marks(score):
    r'''Applies expressive marks to score.
```

```
    '''

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

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

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

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

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

    strings_staff_group = score['Strings Staff Group']
    for voice in iterate(strings_staff_group).by_class(scoretools.Voice):
        markup = markuptools.Markup(r'\italic { (non dim.) }', Down)
        attach(markup, voice[102][0])
```

We use the *indicatortools.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):
    r'''Applies page breaks to 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:
```

```
        command = indicatortools.LilyPondCommand('break', 'after')
        attach(command, bell_voice[measure_index])
```

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

```
def apply_rehearsal_marks(score):
    r'''Applies rehearsal marks to 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:
        command = indicatortools.LilyPondCommand(r'mark \default', 'before')
        attach(command, bell_voice[measure_index])
```

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

```
def apply_final_bar_lines(score):
    r'''Applies final bar lines to score.
    '''

    for voice in iterate(score).by_class(scoretools.Voice):
        bar_line = indicatortools.BarLine('|.')
        attach(bar_line, voice[-1])
```

10.6 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):
    r'''Configures score.
    '''

    spacing_vector = layouttools.make_spacing_vector(0, 0, 8, 0)
    override(score).vertical_axis_group.staff_staff_spacing = spacing_vector
    override(score).staff_grouper.staff_staff_spacing = spacing_vector
    override(score).staff_symbol.thickness = 0.5
    contextualize(score).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):
    r'''Configures LilyPond file.
    '''

    lilypond_file.global_staff_size = 8

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

```
slash_separator = indicatortools.LilyPondCommand('slashSeparator')
lilypond_file.paper_block.system_separator_markup = slash_separator

bottom_margin = lilypondfiletools.LilyPondDimension(0.5, 'in')
lilypond_file.paper_block.bottom_margin = bottom_margin

top_margin = lilypondfiletools.LilyPondDimension(0.5, 'in')
lilypond_file.paper_block.top_margin = top_margin

left_margin = lilypondfiletools.LilyPondDimension(0.75, 'in')
lilypond_file.paper_block.left_margin = left_margin

right_margin = lilypondfiletools.LilyPondDimension(0.5, 'in')
lilypond_file.paper_block.right_margin = right_margin

paper_width = lilypondfiletools.LilyPondDimension(5.25, 'in')
lilypond_file.paper_block.paper_width = paper_width

paper_height = lilypondfiletools.LilyPondDimension(7.25, 'in')
lilypond_file.paper_block.paper_height = paper_height

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

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)
```


Cantus in Memory of Benjamin Britten (1980)
Arvo Pärt
Campana in La
= 112 – 120
ppp
Camp.
div.
con sord.
Vl. I
Vl. II
Va.
Vc.
Vb.
sim.
pp
sole
p
mp


2
17
Camp.
3
p
Vl. I
Vl. II
p
Va.
mp
Vc.
Vb.


22
Camp.
4
Vl. I
p
Vl. II
mp
Va.
Vc.
mp
Vb.

Part IV

Tutorials

CHAPTER

ELEVEN

FIRST STEPS WITH PYTHON, LILYPOND AND ABJAD

11.1 Getting started

Abjad makes powerful programming techniques available to you when you compose. Read through the points below and then click next to proceed.

11.1.1 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.

11.1.2 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.

11.1.3 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.

11.1.4 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.

11.2 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.

11.2.1 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.
```

11.2.2 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
```

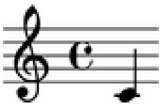
LilyPond reads `hello_world.ly` as input and creates `hello_world.pdf` as output.

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.

11.2.3 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.

11.3 Python "hello, world!" (at the interpreter)

Working with Abjad means programming in Python. Let's start with Python's interactive interpreter.

11.3.1 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.

11.3.2 Entering commands

Type the following at the interpreter's prompt:

```
>>> print 'hello, world!'
hello, world!
```

Python responds by printing `hello, world!` to the terminal.

11.3.3 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.

11.4 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.

11.4.1 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.

11.4.2 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.

11.4.3 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.

11.5 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.

11.5.1 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.

11.5.2 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.

11.6 Abjad "hello, world" (at the interpreter)

11.6.1 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.

11.6.2 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)
```

11.6.3 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.

11.7 Abjad "hello, world!" (in a file)

11.7.1 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.

11.7.2 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.

11.7.3 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.

11.8 More about Abjad

11.8.1 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()
['abjad_configuration', 'Chord', 'Container', 'Duration', 'Fraction',
'Measure', 'Note', 'Rest', 'Score', 'Staff', 'Tuplet', 'Voice',
'__builtins__', '__doc__', '__name__', '__package__',
'__warningregistry__', 'abctools', 'abjadbooktools', 'beamtools',
'scoretools', 'scoretools', 'configurationtools', 'scoretools',
'indicatortools', 'datastructuretools', 'decoratortools',
'developerscripttools', 'documentationtools', 'durationtools',
'exceptiontools', 'f', 'formattools', 'gracetools', 'systemtools',
'instrumenttools', 'introspectiontools', 'systemtools', 'iterationtools',
'labeltools', 'layouttools', 'scoretools', 'lilypondfiletools',
'lilypondparsertools', 'lilypondnametools', 'indicatortools', 'markuptools',
'mathtools', 'scoretools', 'scoretools', 'systemtools', 'p',
'pitcharraytools', 'pitchtools', 'play', 'scoretools', 'rhythmtreetools',
'schemetools', 'templatetools', 'scoretools', 'sequencetools', 'show',
'sievetools', 'scoretools', 'spannertools', 'scoretools', 'stringtools',
'tempotools', 'tietools', 'timeintervaltools', 'metertools',
'rhythmmakertools', 'tonalanalysistools', 'scoretools',
'verticalitytools', 'scoretools', '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.

11.8.2 Inspecting output

Use `dir()` to take a look at the contents of the `systemtools` package:

```
>>> dir(systemtools)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'__path__', '_documentation_section', 'clear_terminal', 'f',
'get_last_output_file_name', 'get_next_output_file_name', 'systemtools',
'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 `systemtools` package implements I/O functions that help you work with the files you create in Abjad.

Use `systemtools.ly()` to see the last LilyPond input file created in Abjad:

```
% Abjad revision 12452
% 2013-10-22 13:32

\version "2.17.3"
\language "english"

\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 `systemtools.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

TWELVE

WORKING WITH NOTATION

12.1 Working with lists of numbers

Python provides a built-in `list` type that you can use to carry around almost anything.

12.1.1 Creating lists

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]
```

12.1.2 Inspecting list attributes

Use `len()` to find the number of elements in any list

```
>>> len(my_list)
12
```

12.1.3 Adding and removing elements

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]
```

12.1.4 Indexing and slicing lists

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]
```

12.1.5 Reversing the order of elements

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]
```

More information on these and all other operations defined on the built-in Python `list` is available in the Python tutorial.

12.2 Changing notes to rests

12.2.1 Making a repeating pattern of notes

It is easy to make a repeating pattern of notes.

Multiplying the list `[0, 2, 4, 9, 7]` by `4` creates a new list of twenty pitch numbers.

The call to `scoretools.make_notes()` creates our notes:

```
>>> pitch_numbers = 4 * [0, 2, 4, 9, 7]
>>> duration = Duration(1, 8)
>>> notes = scoretools.make_notes(pitch_numbers, duration)
>>> staff = Staff(notes)
>>> show(staff)
```

12.2.2 Iterating the notes in a staff

Use `iterate()` to iterate the notes in any expression:

```
>>> for note in iterate(staff).by_class(Note):
...     note
...
Note("c'8")
Note("d'8")
Note("e'8")
Note("a'8")
Note("g'8")
Note("c'8")
Note("d'8")
Note("e'8")
Note("a'8")
Note("g'8")
Note("c'8")
Note("d'8")
Note("e'8")
Note("a'8")
Note("g'8")
Note("c'8")
Note("d'8")
Note("e'8")
Note("a'8")
Note("g'8")
```

12.2.3 Enumerating the notes in a staff

Use Python's built-in `enumerate()` function to enumerate the elements in any iterable:

```
>>> generator = iterate(staff).by_class(Note)
>>> for i, note in enumerate(generator):
...     i, note
...
(0, Note("c'8"))
(1, Note("d'8"))
(2, Note("e'8"))
(3, Note("a'8"))
(4, Note("g'8"))
(5, Note("c'8"))
(6, Note("d'8"))
(7, Note("e'8"))
(8, Note("a'8"))
(9, Note("g'8"))
(10, Note("c'8"))
(11, Note("d'8"))
(12, Note("e'8"))
(13, Note("a'8"))
(14, Note("g'8"))
(15, Note("c'8"))
(16, Note("d'8"))
(17, Note("e'8"))
(18, Note("a'8"))
(19, Note("g'8"))
```

12.2.4 Changing notes to rests by index

We can change every sixth note in a our score to a rest like this:

```
>>> generator = iterate(staff).by_class(Note)
>>> for i, note in enumerate(generator):
...     if i % 6 == 5:
...         rest = Rest('r8')
...         staff[i] = rest
...
```

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

12.2.5 Changing notes to rests by pitch

Let's make a new staff:

```
>>> pitch_numbers = 4 * [0, 2, 4, 9, 7]
>>> duration = Duration(1, 8)
>>> notes = scoretools.make_notes(pitch_numbers, duration)
>>> staff = Staff(notes)
>>> show(staff)
```

Now we can change every D4 to a rest like this:

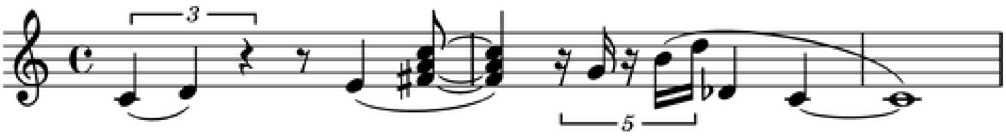
```
>>> generator = iterate(staff).by_class(Note)
>>> for i, note in enumerate(generator):
...     if inspect(note).get_sounding_pitch == "d'":
...         rest = Rest('r8')
```

```
...         staff[i] = rest
...
```

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

12.3 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.



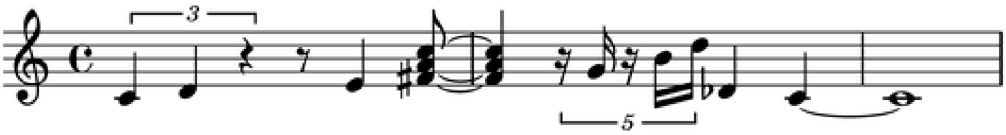
The pattern? Slurs in the example span groups of notes and chords separated by rests.

Abjad makes it easy to create rest-delimited slurs in a structured way.

12.3.1 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)
```



12.3.2 Grouping notes and chords

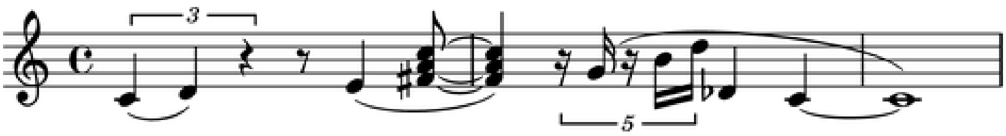
Next we'll group notes and chords together with one of the functions available in the `scoretools` package.

We add slur spanners inside our loop:

```
>>> leaves = iterate(staff).by_class(scoretools.Leaf)
>>> for group in iterate(leaves).by_run((Note, Chord)):
...     slur = Slur()
...     attach(slur, group)
...
```

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.

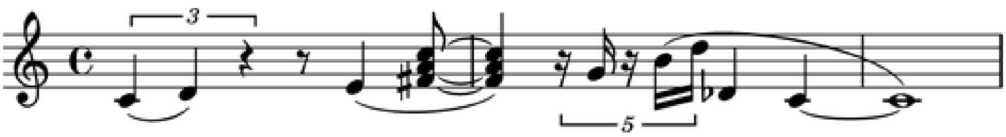
12.3.3 Skipping one-note slurs

Let's rewrite our example to prevent that from happening:

```
>>> staff = Staff(string)
>>> leaves = iterate(staff).by_class(scoretools.Leaf)
>>> for group in iterate(leaves).by_run((Note, Chord)):
...     if 1 < len(group):
...         slur = Slur()
...         attach(slur, group)
...
```

And here's the corrected result:

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



12.4 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.

12.4.1 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 *scoretools.make_notes()*:

```
>>> notes = scoretools.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

12.5 Overriding LilyPond grobs

LilyPond models music notation as a collection of graphic objects or grobs.

12.5.1 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.

12.5.2 Abjad grob-override component plug-ins

Abjad lets you work with LilyPond grobs.

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

```
>>> override(staff).staff_symbol.color = 'blue'
```

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

All Abjad leaves have a grob-override plug-in, too:

```
>>> leaf = staff[-1]
```

```
>>> override(leaf).note_head.color = 'red'
>>> override(leaf).stem.color = 'red'
```

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

And so do Abjad spanners:

```
>>> slur = Slur()
>>> attach(slur, staff[:])
>>> override(slur).slur.color = 'red'
```

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

12.5.3 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()
>>> override(staff).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.

12.5.4 Check the LilyPond docs

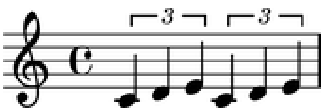
New grobs are added to LilyPond from time to time.

For a complete list of LilyPond grobs see the LilyPond documentation.

12.6 Working with component parentage

Many score objects contain other score objects.

```
>>> tuplet = Tuplet(Multiplier(2, 3), "c'4 d'4 e'4")
>>> staff = Staff(2 * tuplet)
>>> score = Score([staff])
>>> show(score)
```



Abjad uses the idea of parentage to model the way objects contain each other.

12.6.1 Getting the parentage of a component

Use the inspector to get the parentage of any component:

```
>>> note = score.select_leaves()[0]
>>> parentage = inspect(note).get_parentage()
```

```
>>> parentage
Parentage(Note("c'4"), Tuplet(Multiplier(2, 3), "c'4 d'4 e'4"), Staff{2}, Score<<1>>)
```

Abjad returns a special type of selection.

12.6.2 Parentage attributes

Use parentage to find the immediate parent of a component:

```
>>> parentage.parent
Tuplet(Multiplier(2, 3), "c'4 d'4 e'4")
```

Or the root of the score in the which the component resides:

```
>>> parentage.root
Score<<1>>
```

Or to find the depth at which the component is embedded in its score:

```
>>> parentage.depth
3
```

Or the number of tuplets in which the component is nested:

```
>>> parentage.tuplet_depth
1
```

12.7 Working with logical voices

12.7.1 What is a logical voice?

A logical voice is a structural relationship. Abjad uses the concept of the logical voice to bind together all the notes, rests, chords and tuplets that comprise a single musical voice.

It's important to understand what logical voices are and how they impact the way that you may group notes, rests and chords together with beams, slurs and other spanners.

12.7.2 Logical voices vs. explicit voices

Logical voices and explicit voices are different things. The staff below contains an explicit voice. You can slur these notes together because notes contained in an explicit voice always belong to the same logical voice:

```
>>> voice = Voice("c'8 d'8 e'8 f'8")
>>> staff = Staff([voice])
>>> notes = voice.select_leaves()
>>> slur = Slur()
>>> attach(slur, notes)
>>> show(staff)
```

Here is a staff without an explicit voice. You can slur these notes together because both Abjad and LilyPond recognize that the notes belong to the same logical voice even though no explicit voice is present:

```
>>> staff = Staff("g'4 fs'8 e'8")
>>> notes = staff.select_leaves()
>>> slur = Slur()
>>> attach(slur, notes)
>>> show(staff)
```

12.7.3 Different voice names determine different logical voices

Now let’s consider a slightly more complex example. The staff below contains two short voices written one after the other. It’s unusual to think of musical voices as following one after the other on the same staff. But the example keeps things simple while we explore the way that the names of explicit voices impact Abjad’s determination of logical voices:

```
>>> voice_1 = Voice("c’16 d’16 e’16 f’16", name=’First Short Voice’)
>>> voice_2 = Voice("e’8 d’8", name=’Second Short Voice’)
>>> staff = Staff([voice_1, voice_2])
>>> show(staff)
```

You can’t tell that the score above comprises two voices from the notation alone. But the LilyPond input makes this clear:

```
>>> f(staff)
\new Staff {
    \context Voice = "First Short Voice" {
        c’16
        d’16
        e’16
        f’16
    }
    \context Voice = "Second Short Voice" {
        e’8
        d’8
    }
}
```

You can slur together the notes in the first voice:

```
>>> notes = voice_1.select_leaves()
>>> slur = Slur()
>>> attach(slur, notes)
>>> show(staff)
```

And you can slur together the notes in the second voice:

```
>>> notes = voice_2.select_leaves()
>>> slur = Slur()
>>> attach(slur, notes)
>>> show(staff)
```

But you can not slur together all the notes in the staff.

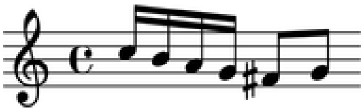
Why? Because the six notes in the staff above belong to two different logical voices. Abjad will raise an exception if you try to slur these notes together. And LilyPond would refuse to render the resulting input code even if you could.

The important point here is that explicit voices carrying different names determine different logical voices. The practical upshot of this is that voice naming constrains which notes, rests and chords you can group together with slurs, beams and other spanners.

12.7.4 Identical voice names determine a single logical voice

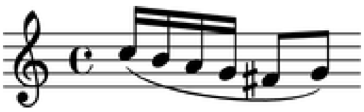
Now let's consider an example in which both voices carry the same name:

```
>>> voice_1 = Voice("c''16 b'16 a'16 g'16", name='Unified Voice')
>>> voice_2 = Voice("fs'8 g'8", name='Unified Voice')
>>> staff = Staff([voice_1, voice_2])
>>> show(staff)
```



All six notes in the staff now belong to the same logical voice. We can see that this is the case because it's now possible to slur all six notes together:

```
>>> voice_1_notes = voice_1.select_leaves()
>>> voice_2_notes = voice_2.select_leaves()
>>> all_notes = voice_1_notes + voice_2_notes
>>> slur = Slur()
>>> attach(slur, all_notes)
>>> show(staff)
```



We can say that this example comprises two explicit voices but only a single logical voice. The LilyPond input code also makes this clear:

```
>>> f(staff)
\new Staff {
    \context Voice = "Unified Voice" {
        c''16 (
        b'16
        a'16
        g'16
    }
    \context Voice = "Unified Voice" {
        fs'8
        g'8 )
    }
}
```

12.7.5 The importance of naming voices

What happens if we choose not to name the explicit voices we create? It is clear that the staff below contains two explicit voices. But because the explicit voices are unnamed it isn't clear how many logical voices the staff defines. Do the notes below belong to one logical voice or two?

```
>>> voice_1 = Voice("c'8 e'16 fs'16")
>>> voice_2 = Voice("g'16 gs'16 a'16 as'16")
>>> staff = Staff([voice_1, voice_2])
>>> show(staff)
```

Abjad defers to LilyPond in answering this question. LilyPond interprets successive unnamed voices as constituting different voices; Abjad follows this convention. This means that you can slur together the notes in the first voice. And you can slur together the notes in the second voice. But you can't slur together all of the notes at once:

```
>>> voice_1_notes = voice_1.select_leaves()
>>> slur = Slur()
>>> attach(slur, voice_1_notes)
>>> voice_2_notes = voice_2.select_leaves()
>>> slur = Slur()
>>> attach(slur, voice_2_notes)
>>> show(staff)
```

This point can be something of a gotcha. If you start working with increasingly fancy ways of structuring your scores you can easily forget that notes in two successive (but unnamed) voices can not be beamed or slurred together.

This leads to a best practice when working with Abjad: **name the explicit voices you create**. The small score snippets we've created for the docs don't really require that names for voices, staves and scores. But scores used to model serious music should provide explicit names for every context from the beginning.

Part V

Reference manual

CHAPTER

THIRTEEN

LEAVES

13.1 Chords

13.1.1 Making chords from a LilyPond input string

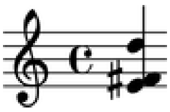
You can make chords from a LilyPond input string:

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

13.1.2 Making chords from numbers

You can also make chords from numbers:

```
>>> chord = Chord([4, 6, 14], Duration(1, 4))
>>> show(chord)
```



13.1.3 Understanding the interpreter representation of a chord

```
>>> chord
Chord("<e' fs' d''>4")
```

`Chord` tells you the chord's class.

`"<e' fs' d''>4"` tells you chord's LilyPond input string.

13.1.4 Getting and setting the written duration of a chord

Get the written duration of a chord like this:

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

Set the written duration of a chord like this:

```
>>> chord.written_duration = Duration(3, 16)
>>> show(chord)
```

13.1.5 Getting and setting the written pitches of a chord

Get the written pitches of a chord like this:

```
>>> chord.written_pitches
(NamedPitch("e'"), NamedPitch("fs'"), NamedPitch("d''"))
```

Set the written pitches of a chord like this:

```
>>> chord.written_pitches = ("e'", "fs'", "gs'")
>>> show(chord)
```

13.1.6 Getting chord note heads

Get the note heads of a chord like this:

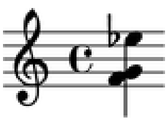
```
>>> for note_head in chord.note_heads: note_head
...
NoteHead("e'")
NoteHead("fs'")
NoteHead("gs'")
```

13.1.7 Appending note heads to a chord

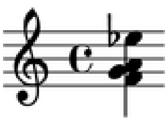
Use `append()` to add one note head to a chord.

You can append with a pitch name:

```
>>> chord = Chord("<f' g' ef''>4")
>>> show(chord)
```



```
>>> chord.note_heads.append("a'")
>>> show(chord)
```



Or with a pitch number:

```
>>> chord.note_heads.append(10)
>>> show(chord)
```

13.1.8 Extending chords

Use `extend()` to add multiple note heads to a chord.

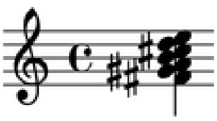
You can extend with pitch names:

```
>>> chord = Chord("<fs' gs' e''>4")
>>> show(chord)
```

```
>>> chord.note_heads.extend(["a'", "b'"])
>>> show(chord)
```

Or with pitch numbers:

```
>>> chord.note_heads.extend([13, 14])
>>> show(chord)
```



13.1.9 Deleting chord note heads

Delete chord note heads with `del()`.

```
>>> chord = Chord("<g' a' f''>4")
>>> show(chord)
```

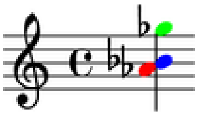
```
>>> del(chord.note_heads[-1])
>>> show(chord)
```

13.1.10 Tweaking chord note heads

Tweak chord note heads like this:

```
>>> chord = Chord("<af' bf' gf''>4")
>>> show(chord)
```

```
>>> chord.note_heads[0].tweak.color = 'red'
>>> chord.note_heads[1].tweak.color = 'blue'
>>> chord.note_heads[2].tweak.color = 'green'
>>> show(chord)
```



13.1.11 Working with empty chords

Abjad allows empty chords:

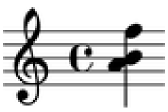
```
>>> chord = Chord([], Duration(1, 4))
>>> chord
Chord('<>4')
```

Empty chords don't constitute valid LilyPond input.

This means LilyPond will complain if you pass empty chords to `show()`.

You can add pitches back to an empty chord at any time:

```
>>> chord.note_heads.extend([9, 11, 17])
>>> show(chord)
```



13.2 Notes

13.2.1 Making notes from a LilyPond input string

You can make notes from a LilyPond input string:

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

13.2.2 Making notes from numbers

You can also make notes from numbers:

```
>>> note = Note(0, Duration(1, 4))
>>> show(note)
```

13.2.3 Understanding the interpreter representation of a note

```
>>> note
Note("c'4")
```

`Note` tells you the note's class.

`c'` tells you that the note's pitch is equal to middle C.

`4` tells you that the note's duration is equal to a quarter note.

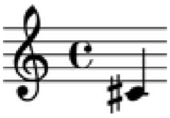
13.2.4 Getting and setting the written pitch of notes

Get the written pitch of notes like this:

```
>>> note.written_pitch
NamedPitch("c'")
```

Set the written pitch of notes like this:

```
>>> note.written_pitch = NamedPitch("cs'")
>>> show(note)
```



Or this:

```
>>> note.written_pitch = "d'"
>>> show(note)
```

Or this:

```
>>> note.written_pitch = 3
>>> show(note)
```

13.2.5 Getting and setting the written duration of notes

Get the written duration of notes like this:

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

Set the written duration of notes like this:

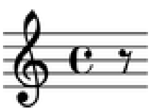
```
>>> note.written_duration = Duration(3, 16)
>>> show(note)
```

13.3 Rests

13.3.1 Making rests from strings

You can make rests from a LilyPond input string:

```
>>> rest = Rest('r8')
>>> show(rest)
```



13.3.2 Making rests from durations

You can make rests from durations:

```
>>> rest = Rest(Duration(1, 4))
>>> show(rest)
```

13.3.3 Making rests from other Abjad leaves

You can also make rests from other Abjad leaves:

```
>>> note = Note("d'4..")
>>> rest = Rest(note)
>>> show(rest)
```

13.3.4 Understanding the interpreter representation of a rest

```
>>> rest
Rest('r4..')
```

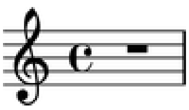
`Rest` tells you the rest's class.

`4..` tells you that the rest's duration is equal to that of a doubly dotted quarter note.

13.3.5 Making multimeasure rests

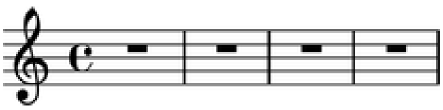
Create multimeasure rests like this:

```
>>> multimeasure_rest = scoretools.MultimeasureRest('R1')
>>> show(multimeasure_rest)
```



Multiply the duration of multimeasure rests like this:

```
>>> attach(Multiplier(4), multimeasure_rest)
>>> staff = Staff([multimeasure_rest])
>>> show(staff)
```



Use a LilyPond command to compress full-bar rests:

```
>>> command = indicatortools.LilyPondCommand('compressFullBarRests')
>>> attach(command, staff)
>>> show(staff)
```

13.3.6 Getting and setting the written duration of rests

Get the written duration of rests like this:

```
>>> rest.written_duration
Duration(7, 16)
```

Set the written duration of rests like this:

```
>>> rest.written_duration = Duration(3, 16)
>>> show(rest)
```

CHAPTER

FOURTEEN

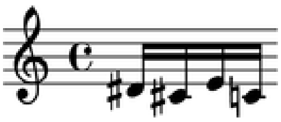
CONTAINERS

14.1 Containers

14.1.1 Creating containers

Create a container with components:

```
>>> notes = [Note("ds'16"), Note("cs'16"), Note("e'16"), Note("c'16")]
>>> container = Container(notes)
>>> show(container)
```



Or with a LilyPond input string:

```
>>> container = Container("ds'16 cs'16 e'16 c'16 d'2 ~ d'8")
>>> show(container)
```

14.1.2 Selecting music

Slice a container to select its components:

```
>>> container[:]
SliceSelection(Note("ds'16"), Note("cs'16"), Note("e'16"), Note("c'16"), Note("d'2"), Note("d'8"))
```

14.1.3 Inspecting length

Get the length of a container with Python's built-in `len()` function:

```
>>> len(container)
6
```

14.1.4 Inspecting duration

Use the inspector the get the duration of a container:

```
>>> inspect(container).get_duration()
Duration(7, 8)
```

14.1.5 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)
```

14.1.6 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)
```

14.1.7 Finding the index of a component

Find the index of a component with `index()`:

```
>>> note = container[7]
```

```
>>> container.index(note)
7
```

14.1.8 Inserting a component by index

Insert a component by index with `insert()`:

```
>>> container.insert(-3, Note("g'32"))
>>> show(container)
```

14.1.9 Removing a component by index

Remove a component by index with `pop()`:

```
>>> container.pop(-1)
Note("a'32")
>>> show(container)
```

14.1.10 Removing a component by reference

Remove a component by reference with `remove()`:

```
>>> container.remove(container[-1])
>>> show(container)
```

14.1.11 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:

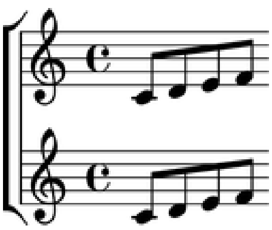
```
>>> f(score)
\new Score <<
    \new StaffGroup <<
        \context Staff = "Flute" {
            c'8
            d'8
            e'8
            f'8
        }
        \context Staff = "Violin" {
            c'8
            d'8
            e'8
            f'8
        }
    >>
>>
```

And make it easy to retrieve containers later:

```
>>> score['Flute']
Staff-"Flute"{4}
```

But container names do not appear in notational output:

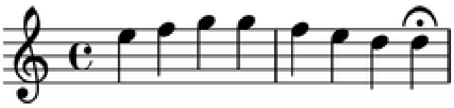
```
>>> show(score)
```



14.1.12 Understanding { } and << >> in LilyPond

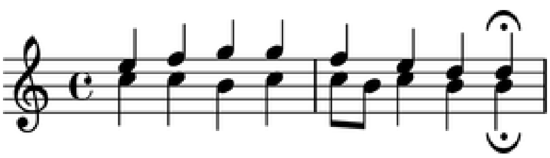
LilyPond uses curly `{ }` braces to wrap a stream of musical events that are to be engraved one after the other:

```
\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.

14.1.13 Understanding sequential and simultneous containers

Abjad implements LilyPond `{ }` and `<< >>` in the container `is_simultaneous` attribute.

Some containers set `is_simultaneous` to false at initialization:

```
>>> staff = Staff([])
>>> staff.is_simultaneous
False
```

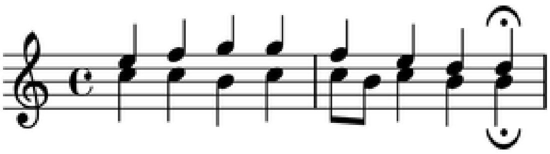
Other containers set `is_simultaneous` to true:

```
>>> score = Score([])
>>> score.is_simultaneous
True
```

14.1.14 Changing sequential and simultaneous containers

Set `is_simultaneous` 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_simultaneous = True
>>> command = indicatortools.LilyPondCommand('voiceOne')
>>> attach(command, voice_1)
>>> command = indicatortools.LilyPondCommand('voiceTwo')
>>> attach(command, voice_2)
>>> show(staff)
```



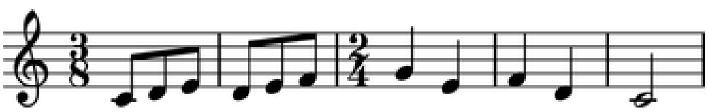
The staff in the example above is set to simultaneous after initialization to create a type of polyphonic staff.

14.2 Measures

14.2.1 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
}
```

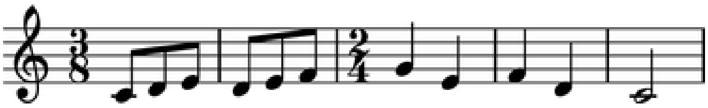


Here LilyPond creates five measures from two time signatures. This happens because behind-the-scenes LilyPond time-keeping tells the program when measures start and stop and how to draw the barlines that come between them.

14.2.2 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")
>>> time_signature_1 = indicatortools.TimeSignature((3, 8))
>>> time_signature_2 = indicatortools.TimeSignature((2, 4))
>>> attach(time_signature_1, staff)
>>> attach(time_signature_2, staff[6])
>>> show(staff)
```



But you may also include explicit measures in the Abjad scores you build. The following sections explain how.

14.2.3 Creating measures

Create a measure with a time signature and music:

```
>>> measure = Measure(TimeSignature((3, 8)), "c'8 d'8 e'8")
>>> show(measure)
```

14.3 Scores

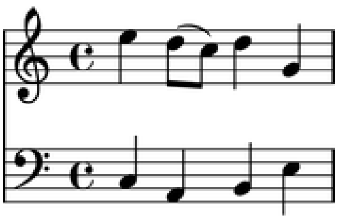
14.3.1 Making a score from a LilyPond input string

You can make an Abjad score from a LilyPond input string:

```
>>> input = r'''
... \new Staff { e''4 d''8 ( c''8 ) d''4 g'4 }
... \new Staff { \clef bass c4 a,4 b,4 e4 }
... '''
```

```
>>> score = Score(input)
```

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



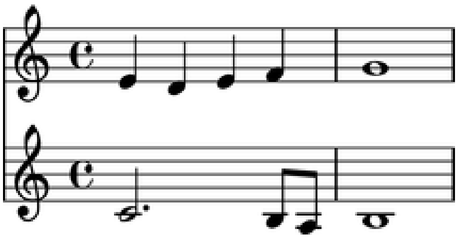
14.3.2 Making a score from a list of Abjad components

You can also make a score from a list of other Abjad components:

```
>>> 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)
```



14.3.3 Understanding the interpreter representation of a score

The interpreter representation of an Abjad score contains three parts:

```
>>> score
Score<<2>>
```

`Score` tells you the score's class.

`2` tells you the score's length (which is the number of top-level components the score contains).

Curly braces `{` and `}` tell you that the music inside the score is interpreted sequentially rather than simultaneously.

14.3.4 Understanding the LilyPond format of a score

Use `format()` to get the LilyPond format of a score:

```
>>> print format(score, 'lilypond')
\new Score <<
    \new Staff {
        e'4
        d'4
        e'4
        f'4
        g'1
    }
    \new Staff {
        c'2.
        b8
        a8
        b1
    }
>>
```

14.3.5 Selecting the music in a score

Slice a score to select its components:

```
>>> score[:]
SimultaneousSelection(Staff{5}, Staff{4})
```

14.3.6 Selecting a score's leaves

Use `select_leaves()` to select the leaves in a score:

```
>>> score.select_leaves(allow_discontiguous_leaves=True)
Selection(Note("e'4"), Note("d'4"), Note("e'4"), Note("f'4"), Note("g'1"), Note("c'2."), Note('b8'), Note('a8'
```

14.3.7 Getting the length of a score

Use `len()` to get the length of a score.

The length of a score is defined equal to the number of top-level components the score contains:

```
>>> len(score)
2
```

14.3.8 Inspecting duration

Use the inspector to get the duration of a score:

```
>>> inspect(score).get_duration()
Duration(2, 1)
```

14.3.9 Appending one component to the bottom of a score

Use `append()` to append one component to the bottom of a score:

```
>>> staff = Staff("g4 f4 e4 d4 d1")
>>> clef = Clef('bass')
>>> attach(clef, staff)
```

```
>>> score.append(staff)
```

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

14.3.10 Finding the index of a score component

Use `index()` to find the index of a score component:

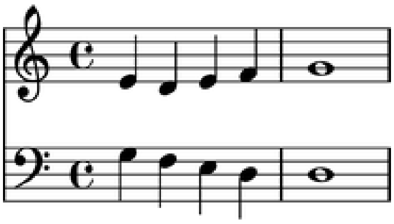
```
>>> score.index(treble_staff_1)
0
```

14.3.11 Removing a score component by index

Use `pop()` to remove a score component by index:

```
>>> score.pop(1)
Staff{4}
```

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

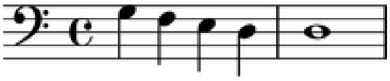


14.3.12 Removing a score component by reference

Use `remove()` to remove a score component by reference:

```
>>> score.remove(treble_staff_1)
```

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



14.3.13 Inspecting whether or not a score contains a component

Use `in` to find out whether a score contains a given component:

```
>>> treble_staff_1 in score
False
```

```
>>> treble_staff_2 in score
False
```

```
>>> staff in score
True
```

14.3.14 Naming scores

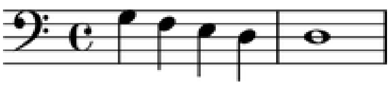
You can name Abjad scores:

```
>>> score.name = 'Example Score'
```

Score names appear in LilyPond input but not in notational output:

```
>>> f(score)
\context Score = "Example Score" <<
    \new Staff {
        \clef "bass"
        g4
        f4
        e4
        d4
        d1
    }
>>
```

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



14.4 Staves

14.4.1 Making a staff from a LilyPond input string

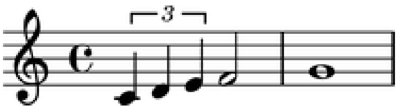
You can make a staff from a LilyPond input string:

```
>>> staff = Staff("c'8 d'8 e'8 f'8 g'8 a'8 b'4 c''1")
>>> show(staff)
```

14.4.2 Making a staff from a list of Abjad components

You can also make a staff from a list of other Abjad components:

```
>>> components = [Tuplet(Multiplier(2, 3), "c'4 d'4 e'4"), Note("f'2"), Note("g'1")]
>>> staff = Staff(components)
>>> show(staff)
```



14.4.3 Understanding the interpreter representation of a staff

The interpreter representation of a staff contains three parts:

```
>>> staff
Staff{3}
```

`Staff` tells you the staff's class.

`3` tells you the staff's length (which is the number of top-level components the staff contains).

Curly braces `{` and `}` tell you that the music inside the staff is interpreted sequentially rather than simultaneously.

14.4.4 Inspecting the LilyPond format of a staff

Use `format()` to get the LilyPond format of a staff:

```
>>> print format(staff, 'lilypond')
\new Staff {
    \times 2/3 {
        c'4
        d'4
        e'4
    }
    f'2
    g'1
}
```

14.4.5 Selecting the music in a staff

Slice a staff to select its components:

```
>>> staff[:]
SliceSelection(Tuplet(Multiplier(2, 3), "c'4 d'4 e'4"), Note("f'2"), Note("g'1"))
```

14.4.6 Selecting a staff's leaves

Use `select_leaves()` to select in the leaves in a staff:

```
>>> staff.select_leaves()
ContiguousSelection(Note("c'4"), Note("d'4"), Note("e'4"), Note("f'2"), Note("g'1"))
```

14.4.7 Getting the length of a staff

Use `len()` to get the length of a staff.

The length of a staff is defined equal to the number of top-level components the staff contains:

```
>>> len(staff)
3
```

14.4.8 Inspecting duration

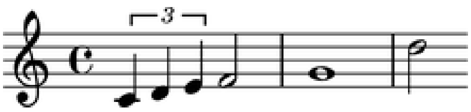
Use the inspector to get the duration of a staff:

```
>>> inspect(staff).get_duration()
Duration(2, 1)
```

14.4.9 Appending one component to the end of a staff

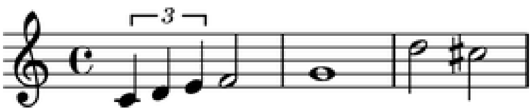
Use `append()` to append one component to the end of a staff:

```
>>> staff.append(Note("d''2"))
>>> show(staff)
```



You can also use a LilyPond input string:

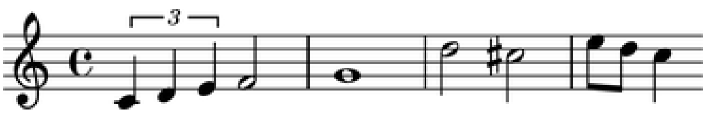
```
>>> staff.append("cs''2")
>>> show(staff)
```



14.4.10 Extending a staff with multiple components at once

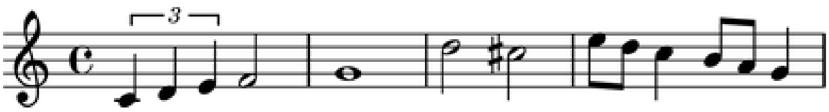
Use `extend()` to extend a staff with multiple components at once:

```
>>> notes = [Note("e''8"), Note("d''8"), Note("c''4")]
>>> staff.extend(notes)
>>> show(staff)
```



You can also use a LilyPond input string:

```
>>> staff.extend("b'8 a'8 g'4")
>>> show(staff)
```



14.4.11 Finding the index of a component in a staff

Use `index()` to find the index of any component in a staff:

```
>>> notes[0]
Note("e''8")
```

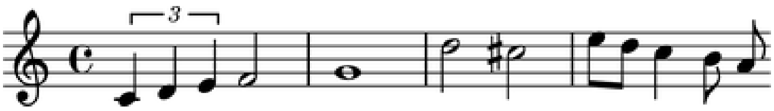
```
>>> staff.index(notes[0])
5
```

14.4.12 Popping a staff component by index

Use `pop()` to pop the last component of a staff:

```
>>> staff[8]
Note("b'8")
```

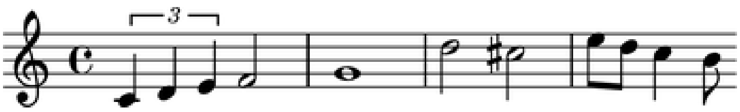
```
>>> staff.pop()
Note("g'4")
>>> show(staff)
```



14.4.13 Removing a staff component by reference

Use `remove()` to remove any component in a staff by reference:

```
>>> staff.remove(staff[-1])
>>> show(staff)
```



14.4.14 Naming staves

You can name Abjad staves:

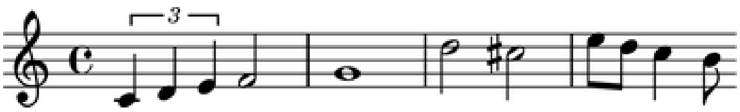
```
>>> staff.name = 'Example Staff'
```

Staff names appear in LilyPond input but not in notational output:

```
>>> f(staff)
\context Staff = "Example Staff" {
    \times 2/3 {
        c'4
        d'4
        e'4
    }
    f'2
    g'1
    d''2
    cs''2
    e''8
    d''8
```

```
    c''4
    b'8
}
```

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



14.4.15 Changing the context of a voice

The context of a staff is set to `Staff` by default:

```
>>> staff.context_name
'Staff'
```

But you can change the context of a staff if you want.

Change the context of a voice when you have defined a new LilyPond context based on a LilyPond staff:

```
>>> staff.context_name = 'CustomUserStaff'
```

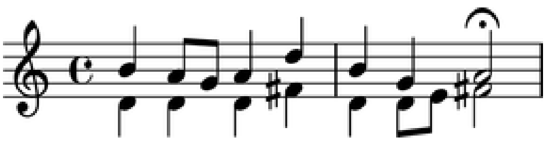
```
>>> staff.context_name
'CustomUserStaff'
```

```
>>> f(staff)
\context CustomUserStaff = "Example Staff" {
    \times 2/3 {
        c'4
        d'4
        e'4
    }
    f'2
    g'1
    d''2
    cs''2
    e''8
    d''8
    c''4
    b'8
}
```

14.4.16 Making parallel voices in a staff

You can make a staff treat its contents as simultaneous with `is_simultaneous`:

```
>>> soprano_voice = Voice(r"b'4 a'8 g'8 a'4 d''4 b'4 g'4 a'2 \fermata")
>>> alto_voice = Voice(r"d'4 d'4 d'4 fs'4 d'4 d'8 e'8 fs'2")
>>> override(soprano_voice).stem.direction = Up
>>> override(alto_voice).stem.direction = Down
>>> staff = Staff([soprano_voice, alto_voice])
>>> staff.is_simultaneous = True
>>> show(staff)
```



14.5 Tuplets

14.5.1 Making a tuplet from a LilyPond input string

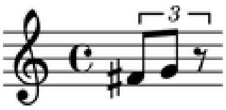
You can tuplets from a LilyPond input string:

```
>>> tuplet = Tuplet(Multiplier(2, 3), "c’8 d’8 e’8")
>>> show(tuplet)
```

14.5.2 Making a tuplet from a list of other components

You can also make tuplets from a list of other components:

```
>>> leaves = [Note("fs’8"), Note("g’8"), Rest(’r8’)]
>>> tuplet = Tuplet(Multiplier(2, 3), leaves)
>>> show(tuplet)
```



14.5.3 Understanding the interpreter representation of a tuplet

The interprer representation of an tuplet contains three parts:

```
>>> tuplet
Tuplet(Multiplier(2, 3), "fs’8 g’8 r8")
```

`Tuplet` tells you the tuplet’s class.

`Multiplier(2, 3)` tells you the tuplet’s multiplier.

`[fs’8, g’8, r8]` tells you the top-level components the tuplet contains.

14.5.4 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 simultaneously.

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.

14.5.5 Formatting tuplets

Use `format()` to get the LilyPond format a tuplet:

```
>>> print format(tuplet, 'lilypond')
\times 2/3 {
    fs'8
    g'8
    r8
}
```

14.5.6 Selecting the music in a tuplet

Select the music in a tuplet like this:

```
>>> tuplet[:]
SliceSelection(Note("fs'8"), Note("g'8"), Rest('r8'))
```

14.5.7 Selecting a tuplet's leaves

Use `select_leaves()` to get the leaves in a tuplet:

```
>>> tuplet.select_leaves()
ContiguousSelection(Note("fs'8"), Note("g'8"), Rest('r8'))
```

14.5.8 Getting the length of a tuplet

Use `len()` to get the length of a tuplet.

The length of a tuplet is defined equal to the number of top-level components the tuplet contains:

```
>>> len(tuplet)
3
```

14.5.9 Inspecting tuplet duration

Use the inspector to get the duration of a tuplet:

```
>>> inspect(tuplet).get_duration()
Duration(1, 4)
```

14.5.10 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
```

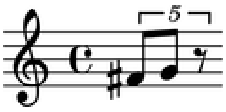
14.5.11 Getting and setting the multiplier of a tuplet

Get the multiplier of a tuplet like this:

```
>>> tuplet.multiplier
Multiplier(2, 3)
```

Set the multiplier of a tuplet like this:

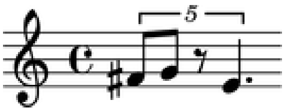
```
>>> tuplet.multiplier = Multiplier(4, 5)
>>> show(tuplet)
```



14.5.12 Appending one component to the end of a tuplet

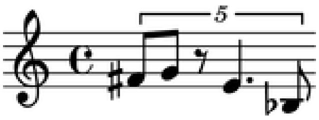
Use `append()` to append one component to the end of a tuplet:

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



You can also use a LilyPond input string:

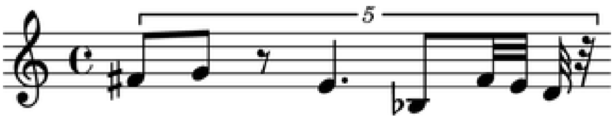
```
>>> tuplet.append("bf8")
>>> show(tuplet)
```



14.5.13 Extending a tuplet with multiple components at once

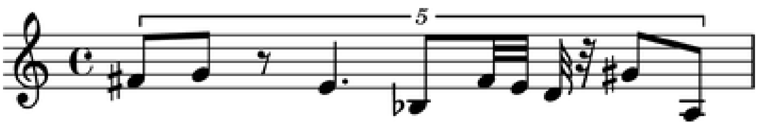
Use `extend()` to extend a tuplet with multiple components at once:

```
>>> notes = [Note("fs'32"), Note("e'32"), Note("d'32"), Rest((1, 32))]
>>> tuplet.extend(notes)
>>> show(tuplet)
```



You can also use a LilyPond input string:

```
>>> tuplet.extend("gs'8 a8")
>>> show(tuplet)
```



14.5.14 Finding the index of a component in a tuplet

Use `index()` to find the index of any component in a tuplet:

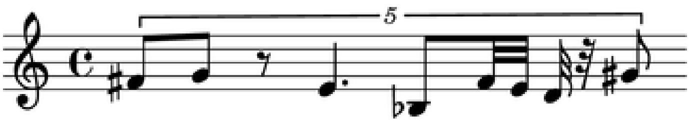
```
>>> notes[1]
Note("e'32")
```

```
>>> tuplet.index(notes[1])
6
```

14.5.15 Popping a tuplet component by index

Use `pop()` to remove the last component of a tuplet:

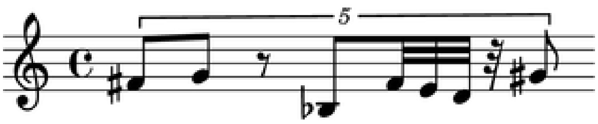
```
>>> tuplet.pop()
Note('a8')
>>> show(tuplet)
```



14.5.16 Removing a tuplet component by reference

Use `remove()` to remove any component from a tuplet by reference:

```
>>> tuplet.remove(tuplet[3])
>>> show(tuplet)
```



14.5.17 Overriding attributes of the LilyPond tuplet number grob

Override attributes of the LilyPond tuplet number grob like this:

```
>>> string = 'tuplet-number::calc-fraction-text'
>>> scheme = schemetools.Scheme(string)
>>> override(tuplet).tuplet_number.text = scheme
>>> override(tuplet).tuplet_number.color = 'red'
>>> staff = Staff([tuplet])
>>> show(staff)
```

See LilyPond's documentation for lists of grob attributes available.

14.5.18 Overriding attributes of the LilyPond tuplet bracket grob

Override attributes of the LilyPond tuplet bracket grob like this:

```
>>> override(tuplet).tuplet_bracket.color = 'red'
>>> show(staff)
```

See LilyPond's documentation for lists of grob attributes available.

14.6 Voices

14.6.1 Making a voice from a LilyPond input string

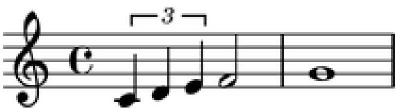
You can make a voice from a LilyPond input string:

```
>>> voice = Voice("c'8 d'8 e'8 f'8 g'8 a'8 b'4 c''1")
>>> show(voice)
```

14.6.2 Making a voice from a list of other components

You can also make a voice from a list of other components:

```
>>> tuplet = Tuplet(Multiplier(2, 3), "c'4 d'4 e'4")
>>> components = [tuplet, Note("f'2"), Note("g'1")]
>>> voice = Voice(components)
>>> show(voice)
```



14.6.3 Understanding the interpreter representation of a voice

The interpreter representation of a 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 simultaneously.

14.6.4 Formatting voices

Use `format()` to get the LilyPond format of a voice:

```
>>> print format(voice, 'lilypond')
\new Voice {
    \times 2/3 {
        c'4
        d'4
        e'4
    }
    f'2
    g'1
}
```

14.6.5 Selecting the components in a voice

Select the components in a voice like this:

```
>>> voice[:]
SliceSelection(Tuplet(Multiplier(2, 3), "c'4 d'4 e'4"), Note("f'2"), Note("g'1"))
```

14.6.6 Selecting a voice's leaves

Use `select_leaves()` to select the leaves in a voice:

```
>>> voice.select_leaves()
ContiguousSelection(Note("c'4"), Note("d'4"), Note("e'4"), Note("f'2"), Note("g'1"))
```

14.6.7 Getting the length of a voice

Use `len()` to get the length of a voice.

The length of a voice is defined equal to the number of top-level components the voice contains:

```
>>> len(voice)
3
```

14.6.8 Inspecting voice duration

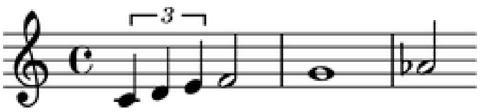
Use the inspector to get the duration of a voice:

```
>>> inspect(voice).get_duration()
Duration(2, 1)
```

14.6.9 Appending one component to the end of a voice

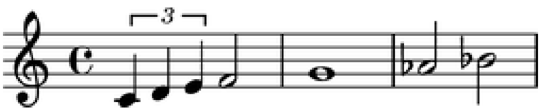
Use `append()` to append one component to the end of a voice:

```
>>> voice.append(Note("af'2"))
>>> show(voice)
```



You can also use a LilyPond input string:

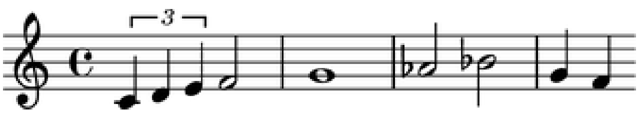
```
>>> voice.append("bf'2")
>>> show(voice)
```



14.6.10 Extending a voice with multiple components at once

Use `extend()` to extend a voice with multiple components at once:

```
>>> notes = [Note("g'4"), Note("f'4")]
>>> voice.extend(notes)
>>> show(voice)
```



You can also use a LilyPond input string:

```
>>> voice.extend("e'4 ef'4")
>>> show(voice)
```

14.6.11 Finding the index of a component in a voice

Use `index()` to find the index of any component in a voice:

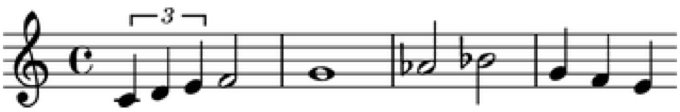
```
>>> notes[0]
Note("g'4")
```

```
>>> voice.index(notes[0])
5
```

14.6.12 Popping a voice component by index

Use `pop()` to pop the last component of a voice:

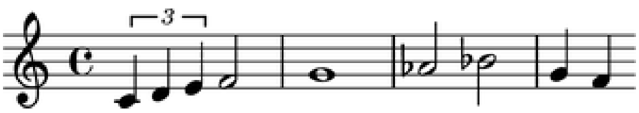
```
>>> voice.pop()
Note("ef'4")
>>> show(voice)
```



14.6.13 Removing a voice component by reference

Use `remove()` to remove any component from a voice by reference:

```
>>> voice.remove(voice[-1])
>>> show(voice)
```



14.6.14 Naming voices

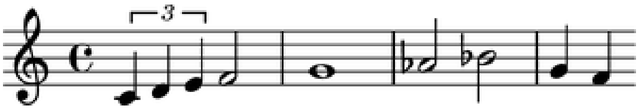
You can name Abjad voices:

```
>>> voice.name = 'Upper Voice'
```

Voice names appear in LilyPond input but not in notation output:

```
>>> f(voice)
\context Voice = "Upper Voice" {
    \times 2/3 {
        c'4
        d'4
        e'4
    }
    f'2
    g'1
    af'2
    bf'2
    g'4
    f'4
}
```

```
>>> show(voice)
```



14.6.15 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.

Change the context of a voice when you have defined a new LilyPond context based on a LilyPond voice:

```
>>> 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
    bf'2
    g'4
    f'4
}
```

CHAPTER

FIFTEEN

LILYPOND FILES

15.1 LilyPond files

15.1.1 Making LilyPond files

Make a basic LilyPond file with the `lilypondfiletools` package:

```
>>> staff = Staff("c'4 d'4 e'4 f'4")
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(staff)
```

```
>>> lilypond_file
LilyPondFile(Staff{4})
```

```
>>> f(lilypond_file)
% 2013-12-18 15:36

\version "2.17.97"
\language "english"

\score {
    \new Staff {
        c'4
        d'4
        e'4
        f'4
    }
}
```

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

15.1.2 Getting header, layout and paper blocks

Basic LilyPond files also come equipped with header, layout and paper blocks:

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

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

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

15.1.3 Setting global staff size and default paper size

Set default LilyPond global staff size and paper size like this:

```
>>> lilypond_file.global_staff_size = 14
>>> lilypond_file.default_paper_size = 'A7', 'portrait'
```

```
>>> f(lilypond_file)
% 2013-12-18 15:36

\version "2.17.97"
\language "english"

#(set-default-paper-size "A7" 'portrait)
#(set-global-staff-size 14)

\score {
    \new Staff {
        c'4
        d'4
        e'4
        f'4
    }
}
```

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

15.1.4 Setting title, subtitle and composer information

Use the LilyPond file header block to set title, subtitle and composer information:

```
>>> lilypond_file.header_block.title = markuptools.Markup('Missa sexti tonus')
>>> lilypond_file.header_block.composer = markuptools.Markup('Josquin')
```

```
>>> f(lilypond_file)
% 2013-12-18 15:36

\version "2.17.97"
\language "english"

#(set-default-paper-size "A7" 'portrait)
#(set-global-staff-size 14)

\header {
    composer = \markup { Josquin }
    title = \markup { Missa sexti tonus }
}

\score {
    \new Staff {
        c'4
        d'4
        e'4
        f'4
    }
}
```

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

Missa sexti tonus

Josquin

Music engraving by LilyPond 2.17.96—www.lilypond.org

CHAPTER

SIXTEEN

ATTACHMENTS

16.1 Annotations

Annotate components with user-specific information.

Annotations do not impact formatting.

16.1.1 Creating annotations

Create annotations like this:

```
>>> annotation_1 = indicatortools.Annotation('is inner voice', True)
```

16.1.2 Attaching annotations to a component

Attach annotations to any component with `attach()`:

```
>>> note = Note("c'4")
>>> attach(annotation_1, note)
```

```
>>> annotation_2 = indicatortools.Annotation('is phrase-initial', False)
>>> attach(annotation_2, note)
```

16.1.3 Getting the annotations attached to a component

Use the inspector to get all the annotations attached to a component:

```
>>> annotations = inspect(note).get_indicators(indicatortools.Annotation)
>>> for annotation in annotations: annotation
...
Annotation('is inner voice', True)
Annotation('is phrase-initial', False)
```

16.1.4 Detaching annotations from a component

Use `detach()` to detach annotations from a component:

```
>>> detach(annotation_1, note)
(Annotation('is inner voice', True),)
```

16.1.5 Inspecting annotation name

Use `name` to get the name of any annotation:

```
>>> annotation_2.name
'is phrase-initial'
```

16.1.6 Inspecting annotation value

Use `value` to get the value of any annotation:

```
>>> annotation_2.value
False
```

16.1.7 Getting the value of an annotation in a single call

Use the inspector to the get the value of an annotation in a single call:

```
>>> inspect(note).get_annotation('is phrase-initial')
False
```

16.2 Articulations

Articulations model staccato dots, marcato wedges and other symbols.

Articulations attach to notes, rests or chords.

16.2.1 Creating articulations

Create articulations like this:

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

16.2.2 Understanding the interpreter representation of an articulation

The interpreter representation of an articulation looks like this:

```
>>> articulation
Articulation('turn')
```

`Articulation` tells you the articulation's class.

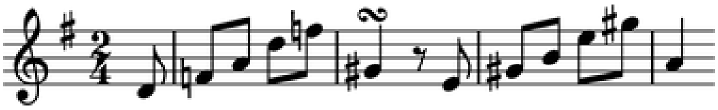
`'staccato'` tells you the articulation's name.

16.2.3 Attaching articulations to a leaf

Use `attach()` to attach articulations to a leaf:

```
>>> staff = Staff()
>>> key_signature = KeySignature('g', 'major')
>>> attach(key_signature, staff)
>>> time_signature = TimeSignature((2, 4), partial=Duration(1, 8))
>>> attach(time_signature, 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")
>>> attach(articulation, staff[5])
```

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

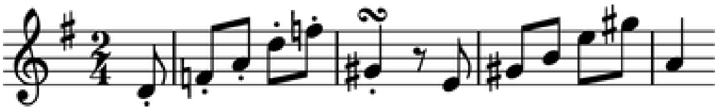


16.2.4 Attaching articulations to many leaves

Write a loop to attach articulations to many leaves:

```
>>> for leaf in staff[:6]:
...     staccato = Articulation('staccato')
...     attach(staccato, leaf)
...
```

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



16.2.5 Getting the articulations attached to a leaf

Use the inspector to get the articulations attached to a leaf:

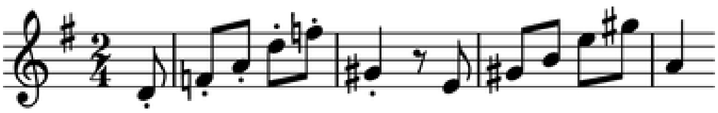
```
>>> inspect(staff[5]).get_indicators(Articulation)
(Articulation('turn'), Articulation('staccato'))
```

16.2.6 Detaching articulations from a leaf

Detach articulations with `detach()`:

```
>>> detach(articulation, staff[5])
(Articulation('turn'),)
```

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



16.2.7 Understanding the string representation of an articulation

The string representation of an articulation comprises two parts:

```
>>> print str(articulation)
-\turn
```

`-` tells you the articulation's direction.

`\staccato` tells you the articulation's LilyPond command.

16.2.8 Understanding the LilyPond format of an articulation

The LilyPond format of an articulation is the same as the articulation's string representation:

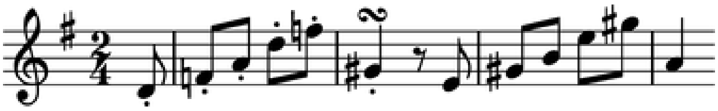
```
>>> print format(articulation, 'lilypond')
-\turn
```

16.2.9 Controlling whether an articulation appears above or below the staff

Use `Up` to force an articulation to appear above the staff:

```
>>> articulation = Articulation('turn', Up)
>>> attach(articulation, staff[5])
```

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

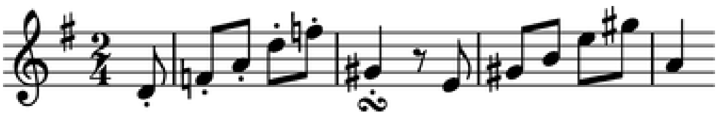


Use `Down` to force an articulation to appear below the staff:

```
>>> detach(articulation, staff[5])
(Articulation('turn', Up),)
```

```
>>> articulation = Articulation('turn', Down)
>>> attach(articulation, staff[5])
```

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



16.2.10 Comparing articulations

Articulations compare equal when name and direction strings compare equal:

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

Otherwise articulations do not compare equal:

```
>>> Articulation('staccato', Up) == Articulation('turn', Up)
False
```

(This chapter's musical examples are based on Haydn's piano sonata number 42, Hob. XVI/27.)

16.3 Instruments

16.3.1 Creating instruments

Use `instrumenttools` to create an instrument:

```
>>> violin = instrumenttools.Violin()
```

16.3.2 Understanding the interpreter representation of an instrument

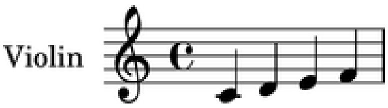
The interpreter representation of an instrument tells you the instrument's class:

```
>>> violin
Violin()
```

16.3.3 Attaching instruments to a component

Use `attach()` to attach an instrument to a component:

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



16.3.4 Inspecting the instrument attached to a component

Use the inspector to get the instrument attached to a component:

```
>>> inspect(staff).get_indicator(instrumenttools.Instrument)
Violin()
```

16.3.5 Inspecting a component's effective instrument

Use the inspector to get the instrument currently in effect for a component:

```
>>> for note in staff:
...     inspect(note).get_effective(instrumenttools.Instrument)
...
Violin()
Violin()
Violin()
Violin()
```

16.3.6 Detaching instruments from a component

Use `detach()` to detach an instrument from a component:

```
>>> detach(violin, staff)
(Violin(),)
>>> show(staff)
```

16.3.7 Getting the name of an instrument

Use `instrument_name` to get the name of any instrument:

```
>>> violin.instrument_name
'violin'
```

Use `instrument_name_markup` to get the instrument name markup of any instrument:

```
>>> violin.instrument_name_markup
Markup(('Violin',))
```

```
>>> show(violin.instrument_name_markup)
```

Violin

16.3.8 Getting the short name of an instrument

Use `short_instrument_name` to get the short name of any instrument:

```
>>> violin.short_instrument_name
'vn.'
```

Use `short_instrument_name_markup` to get the short instrument name markup of any instrument:

```
>>> violin.short_instrument_name_markup
Markup(('Vn.',))
```

```
>>> show(violin.short_instrument_name_markup)
```

Vn.

16.3.9 Getting an instrument's range

Use `pitch_range` to get the range of any instrument:

```
>>> violin.pitch_range
PitchRange('[G3, G7]')
```

```
>>> show(violin.pitch_range)
```

16.3.10 Getting an instrument's level of transposition

Use `sounding_pitch_of_written_middle_c` to get an instrument's level of transposition:

```
>>> violin.sounding_pitch_of_written_middle_c
NamedPitch("c'")
```

```
>>> show(violin.sounding_pitch_of_written_middle_c)
```

16.3.11 Getting an instrument's allowable clefs

Use `allowable_clefs` to get clefs on which an instrument is conventionally notated:

```
>>> violin.allowable_clefs
ClefInventory([Clef('treble')])
```

```
>>> show(violin.allowable_clefs)
```

16.3.12 Customizing instrument properties

You can change the properties of any instrument at initialization:

```
>>> viola = instrumenttools.Viola(
...     instrument_name='Bratsche',
...     short_instrument_name='Br.',
...     allowable_clefs=['alto', 'treble'],
...     pitch_range='[C3, C6]',
...     )
```

```
>>> staff = Staff("c'4 d'4 e'4 fs'4")
>>> attach(viola, staff)
>>> clef = Clef('alto')
>>> attach(clef, staff)
>>> show(staff)
```

16.4 LilyPond commands

LilyPond commands allow you to attach arbitrary LilyPond commands to Abjad score components.

16.4.1 Creating LilyPond commands

Use `indicatortools` to create a LilyPond command:

```
>>> command = indicatortools.LilyPondCommand('bar "||"', 'after')
```

16.4.2 Understanding the interpreter representation of LilyPond commands

```
>>> command
LilyPondCommand('bar "||"', 'after')
```

`LilyPondCommand` tells you the command's class.

`'bar "||"'` tells you the LilyPond command to be formatted.

`'after'` tells you where the command will be formatted relative to the leaf to which it is attached.

16.4.3 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 = KeySignature('f', 'major')
>>> attach(key_signature, staff)
>>> staff.extend("{ d''16 ( c''16 fs''16 g''16 ) }")
>>> staff.extend("{ f''16 ( e''16 d''16 c''16 ) }")
>>> staff.extend("{ cs''16 ( d''16 f''16 d''16 ) }")
>>> staff.extend("{ a'8 b'8 }")
>>> staff.extend("{ d''16 ( c''16 fs''16 g''16 )} ")
>>> staff.extend("{ f''16 ( e''16 d''16 c''16 ) }")
>>> staff.extend("{ cs''16 ( d''16 f''16 d''16 ) }")
>>> staff.extend("{ a'8 b'8 c''2 }")
```

```
>>> attach(command, staff[-2])
```

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

16.4.4 Inspecting the LilyPond commands attached to a leaf

Use the inspector to get the LilyPond commands attached to a leaf:

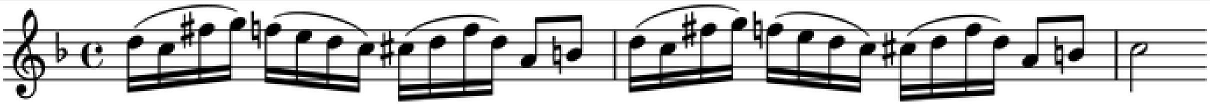
```
>>> inspect(staff[-2]).get_indicators(indicatortools.LilyPondCommand)
(LilyPondCommand('bar "||"', 'after'),)
```

16.4.5 Detaching LilyPond commands

Use `detach()` to detach LilyPond commands:

```
>>> detach(command, staff[-2])
(LilyPondCommand('bar "||"', 'after'),)
```

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



16.4.6 Getting the name of a LilyPond command

Use `name` to get the name of a LilyPond command:

```
>>> command.name
'bar "||"'
```

16.4.7 Comparing LilyPond command marks

LilyPond command marks compare equal with equal names. Otherwise LilyPond command marks do not compare equal:

```
>>> command_1 = indicatortools.LilyPondCommand('bar "||"', 'after')
>>> command_2 = indicatortools.LilyPondCommand('bar "||"', 'before')
>>> command_3 = indicatortools.LilyPondCommand('slurUp')
```

```
>>> command_1 == command_1
True
>>> command_1 == command_2
True
>>> command_1 == command_3
False
```

```
>>> command_2 == command_1
True
>>> command_2 == command_2
True
>>> command_2 == command_3
False
```

```
>>> command_3 == command_1
False
>>> command_3 == command_2
False
>>> command_3 == command_3
True
```

16.5 LilyPond comments

LilyPond comments begin with the `%` sign.

You can include comments in the LilyPond output of the scores you create with Abjad.

16.5.1 Creating LilyPond comments

Use `indicatortools` to create a LilyPond comment:

```
>>> contents_string = 'This is a LilyPond comment before a note.'
>>> comment_1 = indicatortools.LilyPondComment(contents_string, 'before')
```

16.5.2 Understanding the interpreter representation of a LilyPond comment

```
>>> comment_1
LilyPondComment('This is a LilyPond comment before a note.', 'before')
```

`LilyPondComment` tells you the comment's class.

`'This is a LilyPond comments before a note.'` tells you the contents string of the comment.

`'before'` tells you the slot in which the comment will be formatted.

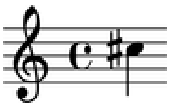
16.5.3 Attaching LilyPond comments to leaves

Use `attach()` to attach LilyPond comments to any note, rest or chord.

You can add LilyPond comments before, after or to the right of any leaf:

```
>>> note = Note("cs''4")
```

```
>>> show(note)
```



```
>>> attach(comment_1, note)
```

```
>>> f(note)
% This is a LilyPond comment before a note.
cs''4
```

16.5.4 Attaching LilyPond comments to containers

Use `attach()` to attach LilyPond comments to a container.

You can add LilyPond comments before, after, in the opening or in the closing of any container:

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

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

```
>>> contents_string_1 = 'Here is a LilyPond comment before the staff.'
>>> contents_string_2 = 'Here is a LilyPond comment in the staff opening.'
>>> contents_string_3 = 'Here is another LilyPond comment in the staff opening.'
>>> contents_string_4 = 'LilyPond comment in the staff closing.'
>>> contents_string_5 = 'LilyPond comment after the staff.'
```

```
>>> staff_comment_1 = indicatortools.LilyPondComment(contents_string_1, 'before')
>>> staff_comment_2 = indicatortools.LilyPondComment(contents_string_2, 'opening')
>>> staff_comment_3 = indicatortools.LilyPondComment(contents_string_3, 'opening')
>>> staff_comment_4 = indicatortools.LilyPondComment(contents_string_4, 'closing')
>>> staff_comment_5 = indicatortools.LilyPondComment(contents_string_5, 'after')
```

```
>>> attach(staff_comment_1, staff)
>>> attach(staff_comment_2, staff)
>>> attach(staff_comment_3, staff)
>>> attach(staff_comment_4, staff)
>>> attach(staff_comment_5, staff)
```

```
>>> 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.
```

16.5.5 Getting the LilyPond comments attached to a component

Use the inspector to get the LilyPond comments attached to any component:

```
>>> inspect(note).get_indicators(indicatortools.LilyPondComment)
(LilyPondComment('This is a LilyPond comment before a note.', 'before'),)
```

16.5.6 Detaching LilyPond comments

Use `detach()` to detach LilyPond comments:

```
>>> detach(comment_1, note)
(LilyPondComment('This is a LilyPond comment before a note.', 'before'),)
```

```
>>> f(note)
cs''4
```

```
>>> detached_comments = detach(indicatortools.LilyPondComment, staff)
>>> for comment in detached_comments: comment
...
LilyPondComment('Here is a LilyPond comment before the staff.', 'before')
LilyPondComment('Here is a LilyPond comment in the staff opening.', 'opening')
LilyPondComment('Here is another LilyPond comment in the staff opening.', 'opening')
LilyPondComment('LilyPond comment in the staff closing.', 'closing')
LilyPondComment('LilyPond comment after the staff.', 'after')
```

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

16.5.7 Getting the contents string of a LilyPond comment

Use `contents_string` to get the contents string of a LiliyPond comment:

```
>>> comment_1.contents_string
'This is a LilyPond comment before a note.'
```

CHAPTER

SEVENTEEN

PITCHES

17.1 Named pitches

Named pitches are the everyday pitches of notes and chords:

```
>>> note = Note("cs''8")
>>> note.written_pitch
NamedPitch("cs''")
```

```
>>> show(note)
```

17.1.1 Creating named pitches

Create named pitches like this:

```
>>> named_pitch = NamedPitch("cs''")
```

17.1.2 Understanding the interpreter representation of a named pitch

```
>>> named_pitch
NamedPitch("cs''")
```

`NamedPitch` tells you the pitch's class.

`cs''` tells you the pitch is equal to `C#5`.

17.1.3 Understanding the string representation of a named pitch

```
>>> str(named_pitch)
"cs''"
```

`cs''` tells you the pitch is equal to `C#5`.

17.1.4 Getting the accidental of a named pitch

Use `accidental` to get the accidental of a named pitch:

```
>>> named_pitch.accidental
Accidental('s')
```

17.1.5 Getting the octave of a named pitch

Use `octave` to get the octave of a named pitch:

```
>>> named_pitch.octave
Octave(5)
```

17.1.6 Comparing named pitches

Named pitches compare equal with equal pitch-class and octave:

```
>>> named_pitch_1 = pitchtools.NamedPitch("cs''")
>>> named_pitch_2 = pitchtools.NamedPitch("df''")
```

```
>>> named_pitch_1 == named_pitch_1
True
>>> named_pitch_1 == named_pitch_2
False
```

```
>>> named_pitch_2 == named_pitch_1
False
>>> named_pitch_2 == named_pitch_2
True
```

You can also compare named pitches with greater-than and less-than:

```
>>> named_pitch_1 < named_pitch_1
False
>>> named_pitch_1 < named_pitch_2
True
>>> named_pitch_2 < named_pitch_1
False
>>> named_pitch_2 < named_pitch_2
False
```

```
>>> named_pitch_1 <= named_pitch_1
True
>>> named_pitch_1 <= named_pitch_2
True
>>> named_pitch_2 <= named_pitch_1
False
>>> named_pitch_2 <= named_pitch_2
True
```

```
>>> named_pitch_1 > named_pitch_1
False
>>> named_pitch_1 > named_pitch_2
False
>>> named_pitch_2 > named_pitch_1
True
>>> named_pitch_2 > named_pitch_2
False
```

```
>>> named_pitch_1 >= named_pitch_1
True
>>> named_pitch_1 >= named_pitch_2
False
>>> named_pitch_2 >= named_pitch_1
True
>>> named_pitch_2 >= named_pitch_2
True
```

17.1.7 Changing named pitches to named pitch-classes

Use `named_pitch_class` to change a named pitch to a named pitch-class:

```
>>> named_pitch.named_pitch_class
NamedPitchClass('cs')
```

Or use `pitchtools`:

```
>>> pitchtools.NamedPitchClass(named_pitch)
NamedPitchClass('cs')
```

17.1.8 Changing named pitches to numbered pitches

Use `numbered_pitch` to change a named pitch to a numbered pitch:

```
>>> named_pitch.numbered_pitch
NumberedPitch(13)
```

Or use `pitchtools`:

```
>>> pitchtools.NumberedPitch(named_pitch)
NumberedPitch(13)
```

17.1.9 Changing named pitches to numbered pitch-classes

Use `numbered_pitch_class` to change a named pitch to a numbered pitch-class:

```
>>> named_pitch.numbered_pitch_class
NumberedPitchClass(1)
```

Or use `pitchtools`:

```
>>> pitchtools.NumberedPitchClass(named_pitch)
NumberedPitchClass(1)
```

Part VI

Developer documentation

CHAPTER

EIGHTEEN

READING AND WRITING CODE

18.1 Codebase

18.1.1 How the Abjad codebase is laid out

The Abjad codebase comprises a small number of top-level directories:

```
abjad$ ls -x -F
__init__.py         __init__.pyc         _version.py         _version.pyc         cfg/
demos/              docs/                etc/                ly/                  scr/
tools/
```

Of these, it is in the `tools` directory that the bulk of the musical reasoning implemented in Abjad resides:

```
abjad$ ls -x -F tools/
__init__.py                 __init__.pyc                abctools/
abjadbooktools/             agenttools/                 configurationtools/
datastructuretools/         developerscripttools/       documentationtools/
durationtools/              exceptiontools/             indicatortools/
instrumenttools/            labeltools/                 layouttools/
lilypondfiletools/          lilypondnametools/          lilypondparsertools/
marktools/                  markuptools/                mathtools/
metertools/                 pitcharraytools/            pitchtools/
quantizationtools/          rhythmmakertools/           rhythmtreetools/
schemetools/                scoretools/                 selectiontools/
sequencetools/              sievetools/                 spannertools/
stringtools/                systemtools/                templatetools/
tempotools/                 test/                       timespantools/
tonalanalysistools/         topleveltools/
```

The remaining sections of this chapter cover the topics necessary to familiarize developers coming to the project for the first time.

18.1.2 Removing prebuilt versions of Abjad before you check out

If you'd like to be at the cutting edge of the Abjad development you first need to check the project out from Google Code, and then teach 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.

If you installed Abjad via `pip`, you can simply say:

```
$ sudo pip uninstall abjad
```

to remove Abjad in one step. We recommend this as the simplest way of installing and uninstalling the packaged version of Abjad. You can download `pip` from https://pypi.python.org/pypi/pip.

If you are unable or uninterested in uninstalling the packaged version of Abjad automatically with `pip`, you'll have to uninstall manually.

To remove prebuilt versions of Abjad resident on your computer manually, you need to find your site packages directory and remove 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 2.7.6
```

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.

18.1.3 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.

18.2 Coding standards

Abjad's coding standards are rigorous, but unambiguous. Code should be written in a clear and consistent manner. This allows not only for long-term legibility, but also facilitates our large collection of codebase tools, which we use to refactor and maintain the system.

We follow PEP8 whenever possible, and our coding standards are quite similar to Google's, which should be considered required reading.

18.2.1 General philosophy

Public is better than private. Explicit is better than implicit. Brevity is almost always acquired along with ambiguity. You're probably only going to type it once, so why make it vaguer than it needs to be? Clarity in purpose and style frees us up to think about more important things... like making music. With that in mind, let's keep our code as clear as possible.

18.2.2 Codebase layout

Avoid private classes.

Avoid private functions. (But use private class methods as necessary.)

Implement only one statement per line of code.

Implement only one class per module.

Implement only one function per module.

18.2.3 Tests

Author one `pytest` test file for every module-level function.

Author one `pytest` test file for every bound method in the public interface of a class.

Author one `doctest` for every public function, method or property.

18.2.4 Casing and naming

Name classes in upper camelcase:

```
def FooBar(object):
    ...
    ...
```

Name bound methods in lower snakecase:

```
def Foo(object):

    def bar_blah(self):
        ...

    def bar_baz(self):
        ...
```

Name module-level functions in lower snakecase:

```
def foo_bar():
    ...

def foo_blah():
    ...
```

Name all variables in lower snakecase:

```
variable_one = 1
variable_two = 2
```

Do not abbreviate variable names, but do use `expr` for 'expression', `i` or `j` for loop counters, and `x` for list comprehensions:

```
def foo(expr):
    result = []
    for i in range(7):
        for j in range(23):
            result.extend(x for x in expr[i][j])
```

Name variables that represent a list or other collection of objects in the plural:

```
some_strings = (
    'one',
    'two',
    'three',
    )
```

Name functions beginning with a verb. (But use `noun_to_noun` for conversion functions and `mathtools.noun` for some `mathtools` functions.)

Preceed private class attributes with a single underscore.

18.2.5 Imports

Avoid `from`. Instead of `from fractions import Fraction` use:

```
import fractions
```

and then qualify the desired classes and functions with the imported module:

```
my_fraction = fractions.Fraction(23, 7)
```

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):
    ...
    ...
```

Use late imports to prevent circular imports problems, especially when importing functionality from within the same tools package.

18.2.6 Whitespace and indentation

Indent with spaces, not with tabs. Use four spaces at a time:

```
def foo(x, y):
    return x + y
```

When enumerating lists, tuples or dictionaries, place each item on its own line, with every item having a trailing comma. Place the final brace on its own line, indented like this:

```
my_tuple = (
    'one',
    'two',
    'three',
    )

my_dictionary = {
    'bar': 2,
    'baz': 3,
    'foo': 1,
    }
```

When a function or method call contains many arguments, prefer to place each argument on its own line as well, with trailing parenthesis:

```
result = my_class.do_something(
    expr,
    keyword_1=True,
    keyword_2=True,
    keyword_3=True,
    )
```

Note: Python (unlike PHP, Java, Javascript etc.) allows for final trailing commas in collections and argument lists. We take advantage of this by placing each item on its own line whenever possible, along with its own trailing comma.

Why? It actually helps us read and write more code.

When adding, subtracting or reordering items in a collection or argument list defined across multiple lines, we never have to think about which item needs to have a comma added, and which needs to have one removed. Similarly, the resulting diffs are much simpler to read. If you keep everything on the same line, the diff will show that the entire line has changed, and you'll have to take time carefully comparing the old and new version to see what (if anything) has been altered. When each item has its own line, the diff will show only the insertion or deletion of a single item.

Use one space around operators:

```
1 + 1
```

instead of:

```
1+1
```

Use no spaces around the = for keyword arguments:

```
my_function(keyword=argument)
```

instead of:

```
my_function(keyword = argument)
```

18.2.7 Line length

Prefer 80 characters whenever possible.

Limit docstring lines to 99 characters.

Limit source lines to 110 characters and use \ to break lines where necessary.

18.2.8 Comments

Introduce comments with one pound sign and a single space:

```
# comment before foo
def foo(x, y):
    return x + y
```

Avoid inline comments.

18.2.9 Docstrings

Wrap docstrings with triple apostrophes and align like this:

```
def foo(x, y):
    r'''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.
    '''
```

Start each docstring with a single sentence explaining, in brief, what the class, function, method or property does.

For class docstrings, and class properties, the article and noun is sufficient, but for methods use a verb, unless that verb is "returns":

```
class NamedPitch(Pitch):
    r'''A named pitch.
    ...
    '''

    ...

    @property
    def accidental(self):
        r'''An accidental.
        ...
```

```
        '''

    ...

    def transpose(self, expr):
        r'''Transpose by `expr`.
        ...
        '''
```

Phrase predicate docstrings like this:

```
class Gesture(object):

    ...

    def is_pitched(self):
        r'''True if gesture is pitched, otherwise false.
        ...
        '''
```

Do not place restructured text double colon *::* symbols at the end of a line of text.

Instead, place all restructured text double colon *::* symbols on lines by themselves, like this:

```
def multiply(x, y):
    r'''Multiplies x by y:

    ::

        >>> foo(10, 11)
        110

    Returns integer.
    '''
```

18.2.10 Quotation

Use paired apostrophes to delimit strings:

```
s = 'foo'
```

Use paired quotation marks to delimit strings within a string:

```
s = 'foo and "bar"'
```

18.2.11 Functions and methods

Alphabetize keyword arguments:

```
my_function(one=1, three=3, two=2)
```

```
my_function(one=1, two=2, three=3)
```

Always include keyword argument names explicitly in function calls:

```
my_function(expr, one=1, three=3, two=2)
```

But not:

```
my_function(expr, 1, 3, 2)
```

Note: Python let's you write out the arguments to a function or method as though they were all positional:

```
def foo(expr, first=None, second=None, third=None):
    ...
```

```
foo(expr, 1, 2, 3)
```

Do *not* do this.

We ask that keyword arguments are always named explicitly because it makes function calls completely unambiguous, and therefore make it easier to refactor using automated tools. In the above function definition, what is our cognitive burden if we realize we need to rename the keyword `third` to `alpha`, but we haven't named the keywords explicitly in our use of the function?

```
def foo(expr, first=None, second=None, alpha=None):
    ...
```

The old function call `foo(expr, 1, 2, 3)` will still work correctly, because we haven't reordered the keywords in the function's signature. But that's burdensome for us, as we're now relying not on the *lexical* ordering of the keyword names, but on their *position*. They might as well be positional arguments. Don't do this! Always explicitly name your keyword arguments, and assume that they can and will be renamed and re-alphabetized at any time. Typing a few extra character is not a burden, but intuiting context while proofreading old code is.

18.2.12 Classes and class file layout

Organize the definitions of classes into the seven following major sections, omitting sections if they contain no class members:

```
class FooBar(object):

    ### CLASS VARIABLES ###

    special_enumeration = (
        'foo',
        'bar',
        'blah',
        )

    ### INITIALIZER ###

    def __init__(self, x, y):
        ...

    ### SPECIAL METHODS ###

    def __repr__(self):
        ...

    def __str__(self):
        ...

    ### PRIVATE PROPERTIES ###

    @apply
    def _bar():
        def fget(self):
            ...
        def fset(self, expr):
            ...
        return property(**locals())

    @property
    def _foo(self):
        ...

    ### PRIVATE METHODS ###

    def _blah(self, x, y):
        ...

    ### PUBLIC PROPERTIES ###
```

```
    @property
    def baz(self):
        ...

    @apply
    def quux():
        def fget(self):
            ...
        def fset(self, expr):
            ...
        return property(**locals())

    ### PUBLIC METHODS ###

    def wux(self, expr, keyword=None):
        ...
```

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):
        ...
```

Alphabetize method names.

18.2.13 Operators

Use < less-than signs in preference to greater-than signs:

```
if x < y < z:
    ...
```

18.2.14 Misc

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()`.

18.3 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.

18.3.1 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 -x -F docs/
Makefile        __init__.py       __init__.pyc       build/              make.bat
pdf/                source/
```

The documentation sourcefiles are collected in section directories resident in `docs/source/`:

```
abjad$ ls -x -F docs/source/
__init__.py                         __init__.pyc
_ext/                                 _static/
_templates/                         _themes/
api/                                 appendices/
conf.py                                 conf.pyc
contents.rst                         developer_documentation/
examples/                         in_conversation/
index.rst                         mothballed/
reference_manual/                 start_here/
system_overview/                 tutorials/
```

The nine section directories in `docs/source` mirror the frontpage sections of the Abjad documentation. There are section directories for the start here, system overview, examples, tutorials, reference manual, developer documentation, appendices, and api and "in conversation" sections of documentation.

When you look inside a section directory you'll find a collection of chaper directories.

Here are the reference manual chapter directories:

```
abjad$ ls -x -F docs/source/reference_manual
annotations/                articulations/                chords/
containers/                images/                        index.rst
instruments/                lilypond_commands/        lilypond_comments/
lilypond_files/                measures/                named_pitches/
notes/                        rests/                        scores/
staves/                        tuplets/                voices/
```

And when you look inside a chapter directory you'll find an `.rst.raw` file, and `.rst` file and an `images/` directory:

```
abjad$ ls -x -F docs/source/reference_manual/notes
images/                index.rst        index.rst.raw
```

18.3.2 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:

```
abjad$ sphinx-build --version
Sphinx (sphinx-build) 1.2
```

If Sphinx responds then the program is already installed on your machine. Otherwise visit the Sphinx website.

18.3.3 Using `ajv api`

The `ajv` application ships with Abjad. The application helps developers manage the Ajbad codebase. The `ajv` subcommand `api` allows for building and cleaning various formats of Sphinx documentation.

```
abjad$ ajv api --help
usage: build-api [-h] [--version] [-M] [-X] [-C] [-O] [--format FORMAT]

Build the Abjad APIs.

optional arguments:
  -h, --help          show this help message and exit
  --version           show program's version number and exit
  -M, --mainline      build the mainline API
```

```
  -X, --experimental  build the experimental API
  -C, --clean         run "make clean" before building the api
  -O, --open          open the docs in a web browser after building
  --format FORMAT     Sphinx builder to use
```

18.3.4 Removing old builds of the documentation

To remove old builds of the documentation, use the `clean` command:

```
abjad$ ajv api --clean
```

18.3.5 Building the HTML docs

You can use `ajv` to build the HTML docs. It doesn't matter what directory you're in when you run the following command:

```
abjad$ ajv api -M
Now writing ReStructured Text files ...

... done.

Now building the HTML docs ...

sphinx-build -b html -d build/doctrees   source build/html
Making output directory...
Running Sphinx v1.1.3
loading pickled environment... not yet created
loading intersphinx inventory from http://docs.python.org/2.7/objects.inv...
building [html]: targets for 1131 source files that are out of date
updating environment: 1131 added, 0 changed, 0 removed
reading sources... [  1%] api/demos/part/PartCantusScoreTemplate/PartCantusScore
reading sources... [  4%] api/tools/abjadbooktools/AbjadBookProcessor/AbjadBookP
reading sources... [  4%] api/tools/abjadbooktools/AbjadBookScript/AbjadBookScri
reading sources... [  4%] api/tools/abjadbooktools/HTMLOutputFormat/HTMLOutputFo
reading sources... [  4%] api/tools/abjadbooktools/LaTeXOutputFormat/LaTeXOutput
reading sources... [  4%] api/tools/abjadbooktools/ReSTOutputFormat/ReSTOutputFo
reading sources... [  5%] api/tools/scoretools/Chord/Chord
...
...
...
copying images... [ 89%] reference_manual/lilypond_commands/images/index-2.
copying images... [ 93%] tutorials/understanding_time_signatures/images/ind
copying images... [ 94%] tutorials/working_with_threads/images/thread-resolution
copying images... [100%] reference_manual/staves/images/index-8.png
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 the `docs/build/html/` directory:

```
abjad$ ls docs/build/
doctrees
latex
```

The output from Sphinx is verbose the first time you build the docs. On sequent builds, Sphinx reports changes only:

```
abjad$ ajv api -M
Now writing ReStructured Text files ...

... done.

Now building the HTML docs ...
```

```
sphinx-build -b html -d build/doctrees   source build/html
Running Sphinx v1.1.3
loading pickled environment... done
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
looking for now-outdated files... none found
no targets are out of date.

Build finished. The HTML pages are in build/html.
```

18.3.6 Building a PDF of the docs

Building a PDF of the docs is almost as simple as building the HTML documentation:

```
abjad$ ajv api -M --format latexpdf
Now writing ReStructured Text files ...

... done.

Now building the LATEXPDF docs ...

sphinx-build -b latex -d build/doctrees   source build/latex
Running Sphinx v1.2b1
loading pickled environment... done
building [latex]: all documents
updating environment: 0 added, 1 changed, 0 removed
reading sources... [100%] developer_documentation/index
looking for now-outdated files... 10 found
pickling environment... done
checking consistency... done
processing Abjad.tex..
...
...
...
Transcript written on AbjadAPI.log.
pdflatex finished; the PDF files are in build/latex.
```

The resulting docs will appear as `Abjad.pdf` and `AbjadAPI.pdf` in the LaTeX build directory, `docs/build/latex`.

18.3.7 Building a coverage report

Build the coverage report with `ajv api` and the `coverage` format.

```
abjad$ ajv api -M --format coverage
Now writing ReStructured Text files ...

... done.

Now building the COVERAGE docs ...

Running Sphinx v1.2b1
loading pickled environment... done
building [coverage]: coverage overview
updating environment: 0 added, 1 changed, 0 removed
reading sources... [100%] api/tools/developerscripttools/BuildApiScript/BuildApiScript
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
```

18.3.8 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
singlehtml to make a single large HTML file
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
devhelp    to make HTML files and a Devhelp project
epub       to make an epub
latex      to make LaTeX files, you can set PAPER=a4 or PAPER=letter
latexpdf   to make LaTeX files and run them through pdflatex
text       to make text files
man        to make manual pages
texinfo    to make Texinfo files
info       to make Texinfo files and run them through makeinfo
gettext    to make PO message catalogs
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)
book       to run abjad-book on all ReST files in source
```

18.3.9 Inserting images with `abjad-book`

Use *ajv 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$ ajv book foo.rst.raw

Parsing file ...
Rendering "example-1.ly" ...
Rendering "example-2.ly" ...
```

You will need to build the HTML docs again to see your work:

```
abjad$ ajv api -M
```

18.3.10 Updating Sphinx

It is important periodically to update your version of Sphinx. If you used `pip` to install Sphinx then the usual command to update Sphinx is this:

```
abjad$ sudo pip install --upgrade Sphinx
```

18.4 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. Abjad 2.13 includes more than 10,000 tests.

18.4.1 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 pytest. The `pytest` 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 `pytest`, and then run the existing battery.

18.4.2 Running the battery

Change to the directory where you have Abjad installed. Then run `pytest`:

```
abjad$ pytest
============================== test session starts ==============================
platform darwin -- Python 2.7.3 -- pytest-2.3.4
collected 4361 items / 3 skipped

demos/desordre/test/test_demos_desordre.py .
demos/ferneyhough/test/test_demos_ferneyhough.py .
demos/mozart/test/test_demos_mozart.py .
demos/part/test/test_demos_part.py .
demos/part/test/test_demos_part_create_pitch_contour_reservoir.py .
demos/part/test/test_demos_part_durate_pitch_contour_reservoir.py .
demos/part/test/test_demos_part_shadow_pitch_contour_reservoir.py .
ly/test/test_ly_environment.py .
tools/abctools/AbjadObject/test/test_AbjadObject___repr__.py ..
tools/scoretools/Chord/test/test_Chord___contains__.py ..
tools/scoretools/Chord/test/test_Chord___copy__.py .....
tools/scoretools/Chord/test/test_Chord___deepcopy__.py .
...
...
...
tools/scoretools/Tuplet/test/test_Tuplet_toggle_prolation.py ..
tools/scoretools/Voice/test/test_Voice___copy__.py ..
tools/scoretools/Voice/test/test_Voice___delitem__.py .
tools/scoretools/Voice/test/test_Voice___len__.py ..
tools/scoretools/Voice/test/test_Voice___setattr__.py .
tools/scoretools/Voice/test/test_Voice_is_nonsemantic.py ...
tools/scoretools/Voice/test/test_lily_voice_resolution.py ....

=================== 4359 passed, 5 skipped in 147.13 seconds ===================
```

Abjad 2.13 includes 4359 `pytest` tests.

18.4.3 Reading test output

`pytest` crawls the entire directory structure from which you call it, running tests in alphabetical order. `pytest` 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.

18.4.4 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.

18.4.5 Test files start with `test_`

When `pytest` first starts up it crawls the entire directory structure from which you call it prior to running a single test. As `pytest` 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 `pytest` begin execution. This collect-and-cache behavior leads to the important point about naming, below.

18.4.6 Avoiding name conflicts

Note that the names of **test functions** must be absolutely unique across the entire directory structure on which you call `pytest`. 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 `pytest` runs. And, unfortunately, **pytest is silent about such conflicts when it runs**.

That is, should you run `pytest` with the duplicate naming situation described here, what will happen is that `pytest` will correctly run and report results for the first such test it finds. However, when `pytest` encounters the second like-named test, `pytest` 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()`.

18.4.7 Updating `pytest`

It is important periodically to update `pytest`.

The usual command to do this is:

```
$ sudo pip install --upgrade pytest
```

Note that `pytest` is here spelled without the intervening period.

18.4.8 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.

You can use the Abjad `ajv` developer suite to run `doctest` anywhere in the codebase:

```
abjad$ ajv doctest
Total modules: 954
```

Output like that shown above indicates that all doctests pass; errors will print to the terminal.

Abjad 2.13 includes more than 7000 doctests.

CHAPTER

NINETEEN

DEVELOPER TOOLS

19.1 Using ajv

Abjad ships with an extensive collection of developer tools. The tools are accessible through the `ajv` developer suite.

You'll find `ajv` in the `abjad/scr/` directory. Make sure to add that directory to your path if you want to work with `ajv`.

The `ajv` developer suite implements a command-line interface that is largely self-documenting:

```
abjad$ ajv --help
usage: abj-dev [-h] [--version]

            {help,list,api,book,clean,count,doctest,grep,new,re,rename,replace,svn,test,up}
            ...

Entry-point to Abjad developer scripts catalog.

optional arguments:
-h, --help            show this help message and exit

subcommands:
{help,list,api,book,clean,count,doctest,grep,new,re,rename,replace,svn,test,up}
    help                print subcommand help
    list                list subcommands
    api                 Build the Abjad APIs.
    book                Preprocess HTML, LaTeX or ReST source with Abjad.
    clean               Clean *.pyc, *.swp, __pycache__ and tmp*
    count               "count"-related subcommands
    doctest             Run doctests on all modules in current path.
    grep                grep PATTERN in PATH
    new                 "new"-related subcommands
    re                  Run pytest -x, doctest -x and then rebuild the API
    rename              Rename public modules.
    replace             "replace"-related subcommands
    svn                 "svn"-related subcommands
    test                Run "pytest" on various Abjad paths.
    up                  run `ajv svn up -R -C`
```

You can explore the different `ajv` subcommands like this:

```
abjad$ ajv clean --help
usage: clean [-h] [--version] [--pyc] [--pycache] [--swp] [--tmp] [path]

Clean *.pyc, *.swp, __pycache__ and tmp* files and folders from PATH.

positional arguments:
path        directory tree to be recursed over

optional arguments:
-h, --help  show this help message and exit
--pyc       delete *.pyc files
--pycache   delete __pycache__ folders
--swp       delete Vim *.swp file
--tmp       delete tmp* folders
```

19.1.1 Searching the Abjad codebase with `ajv grep`

Abjad provides a wrapper around UNIX `grep` in the form of `ajv grep`:

```
$ ajv grep is_assignable
./Duration/Duration.py:361:        if not self.is_assignable:
./Duration/Duration.py:403:        while not candidate.is_assignable:
./Duration/Duration.py:477:        while not candidate.is_assignable:
./Duration/Duration.py:621:    def is_assignable(self):
./Duration/Duration.py:629:            ...         duration.is_assignable)
./Duration/Duration.py:654:                if mathtools.is_assignable_integer(self.numerator):
./Duration/Duration.py:671:        if not self.is_assignable:
```

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 `ajv grep` from any directory on your system; you needn't be in the Abjad source directories when you call `ajv grep`.

Alternatively you may prefer to install `ack` on your system.

19.1.2 Removing old files with `ajv clean`

See the section on `ajv 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 `ajv clean` to delete all the `*.pyc` in the Abjad codebase, leaving other `*.pyc` on your system untouched.

19.1.3 Updating your development copy of Abjad with `ajv up`

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 `ajv up` command as a wrapper around the usual Subversion update commands.

In addition to updating your working copy of Abjad, `ajv up` populates the `abjad/_version.py` 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 `ajv up` takes care of these two situations.

19.1.4 Counting classes and functions with `ajv count`

You can use `ajv count tools .` on the `abjad/tools/` directory to get a count of classes and functions:

```
tools$ ajv count tools .
PUBLIC FUNCTIONS:  465
PUBLIC CLASSES:    486
PRIVATE FUNCTIONS: 38
PRIVATE CLASSES:   0
```

19.1.5 Global search-and-replace with `ajv replace`

You probably won't need to use `ajv replace` 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 `ajv replace` to save lots of time:

```
$ ajv replace text . 'FooBar' 'foo_bar' -Y
```

19.2 Using `ajv book`

`ajv book` is an independent application included in every installation of Abjad. `ajv book` allows you to write Abjad code in the middle of documents written in HTML, LaTeX or ReST. We created `ajv book` to help us document Abjad. Our work on `ajv book` was inspired by `lilypond-book`, which does for LilyPond much what `ajv book` does for Abjad.

`ajv book` can be accessed on the commandline either via `ajv book` or through Abjad's `ajv` tool collection. For the most up-to-date documentation on `ajv book`, always consult `ajv book --help`:

```
abjad$ ajv book --help
usage: abjad-book [-h] [--version] [--skip-rendering] [--verbose] [-X] [-M]
                  [path]

Preprocess HTML, LaTeX or ReST source with Abjad.

positional arguments:
  path                directory tree to be recursed over

optional arguments:
  -h, --help          show this help message and exit
  --version           show program's version number and exit
  --skip-rendering    skip all image rendering and simply execute the code
  --verbose           run in verbose mode, printing all LilyPond output
  -X, --experimental  rebuild abjad.tools docs after processing
  -M, --mainline      rebuild mainline docs after processing

DESCRIPTION

    abjad-book processes Abjad snippets embedded in HTML, LaTeX, or ReST
    documents. All Abjad code placed between the <abjad> </abjad> tags in
    either HTML, LaTeX or ReST type documents is executed and replaced with
    tags appropriate to the given file type. All output generated by the
    code snippet is captured and inserted in the output file.

    Apart from the special opening and closing Abjad tags, abjad-book also
    has a special line-level suffix tag: `<hide`. All lines ending with the
    `<hide` tag will be interpreted by Abjad but will not be displayed in the
    OUTPUT document.

    The opening <abjad> tag can also be followed by a list of
    `attribute=value` pair.

    You can make all of an Abjad code block invisible in the output file with
    the following opening tag:

    <abjad>[hide=true]

    This is useful for generating and embedding rendered score images without
    showing any of the Abjad code.

    You can also remove all of the prompts from a code block with the
    following opening tag:

    <abjad>[strip_prompt=true]

    Simply use Abjad's show() function to have Abjad call LilyPond on the
    Abjad snippet and embed the rendered image in the document.
```

```
    All Abjad snippets *must* start with no indentation in the document.

EXAMPLES

    1.  Create an HTML, LaTex or ReST document with embedded Abjad code
        between <abjad></abjad> tags. The code *must* be fully flushed
        to the left, with no tabs or spaces. The content of an HTML file
        with embedded Abjad might look like this:

        This is an <b>HTML</b> document. Here is Abjad code:

        <abjad>
        voice = Voice("c'4 d'4 e'4 f'4")
        beam = spannertools.Beam()
        attach(beam, voice)
        show(voice)
        </abjad>

        More ordinary <b>HTML</b> text here.

   2.  Call `abjad-book` on the file just created:

        $ abjad-book file.htm.raw
```

19.2.1 HTML with embedded Abjad

To see `ajv 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 = spannertools.Beam(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.

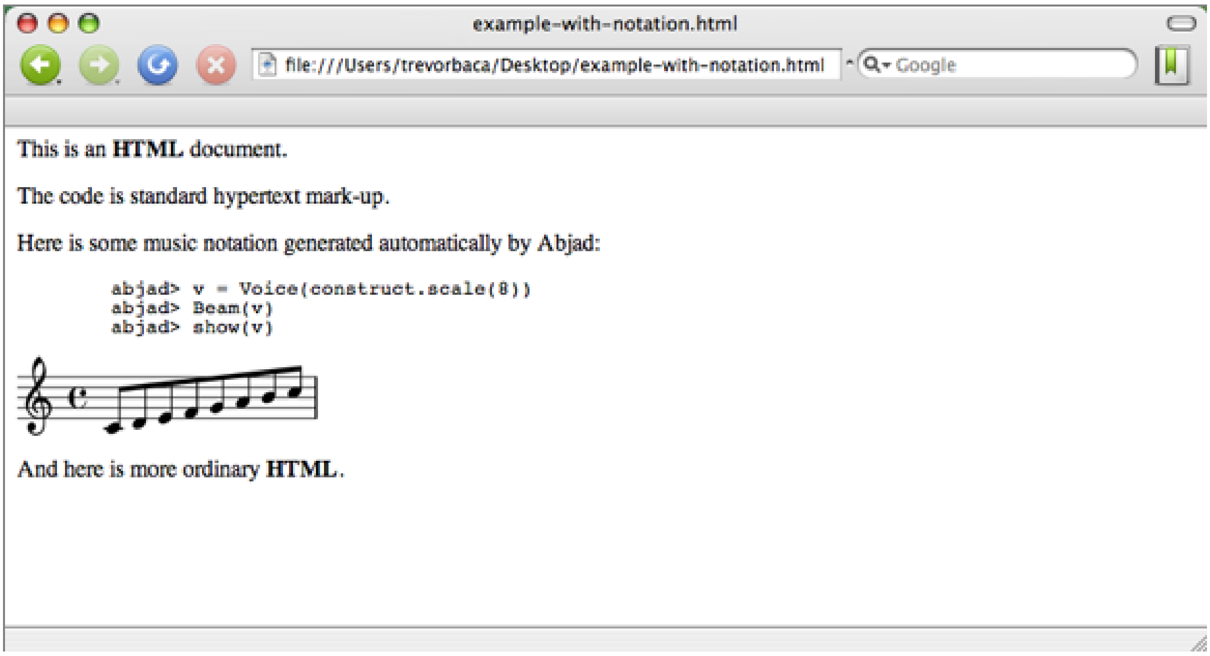
In the terminal, call `ajv book` in the directory:

```
$ ajv book

Parsing file...
Rendering "ajv 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. `ajv book` lets you open a file and type HTML by hand with Abjad sandwiched between the special <abjad> </abjad> tags described here. Run `ajv book` on such a hybrid file to create pure HTML with images of music notation created by Abjad.

Note that `ajv book` makes use of ImageMagick's convert application to crop and scale PNG images generated for HTML and ReST documents. For LaTeX documents, `ajv book` uses `pdfcrop` for cropping PDFs.

19.2.2 LaTeX with embedded Abjad

You can use `ajv 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>
show(measure)
</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 `ajv book` on `example.tex.raw`:

```
$ ajv book example.tex.raw example.tex
```

```
Processing 'example.tex.raw'. Will write output to 'example.tex'...
Parsing file...
Rendering "ajv 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 `ajv 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/ajv 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.

19.2.3 Using `ajv book` on ReST documents

You can call `ajv book` on ReST documents, too. Follow the examples given here for HTML and LaTeX documents and modify accordingly.

19.2.4 Using `[hide=true]`

You can add `[hide=true]` to any `ajv 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")
show(staff)
</abjad>
```

CHAPTER

TWENTY

DEVELOPMENT NOTES

20.1 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)
```

```
0.12424993515
```

These results show that 1000 notes take 0.12 seconds to create.

Other Python timing modules are available for download on the public Internet.

20.2 Profiling code

Profile code with `profile_expr()` in the `systemtools` package:

```
>>> systemtools.IOManager.profile_expr('Note(0, (1, 4))')
Fri Oct 18 14:24:16 2013

        1242 function calls (1121 primitive calls) in 0.003 seconds

Ordered by: cumulative time
List reduced from 83 to 12 due to restriction <12>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.003    0.003 <string>:1(<module>)
     1    0.000    0.000    0.003    0.003 Note.py:45(__init__)
    18    0.000    0.000    0.002    0.000 abc.py:128(__instancecheck__)
    27    0.000    0.000    0.002    0.000 {isinstance}
 68/11    0.001    0.000    0.002    0.000 abc.py:148(__subclasscheck__)
     1    0.000    0.000    0.002    0.002 NoteHead.py:33(__init__)
     1    0.000    0.000    0.002    0.002 NoteHead.py:237(fset)
     1    0.000    0.000    0.002    0.002 NamedPitch.py:29(__init__)
 75/11    0.000    0.000    0.001    0.000 {issubclass}
     1    0.000    0.000    0.001    0.001 Leaf.py:36(__init__)
    85    0.000    0.000    0.001    0.000 _weakrefset.py:58(__iter__)
     1    0.000    0.000    0.000    0.000 NamedPitch.py:232(_initialize_by_pitch_number)
```

These results show 1242 function calls to create a note.

20.3 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.scoretools.Note.Not
     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.NamedPi
                                              icPitch.NamedPitch
     4   1000   9    44000   8     412764  70
                                              abjad.interfaces._OffsetInterf
                                              setInterface
     5   1000   9    44000   8     456764  78 abjad.tools.scoretools.NoteHead
     6   1000   9    40000   7     496764  85 0x23add0
     7   1000   9    32000   5     528764  90
                                              abjad.interfaces.ParentageInte
                                              ParentageInterface
     8   1011   9    28568   5     557332  95 str
     9   1000   9    28000   5     585332 100
                                              abjad.interfaces._NavigationIn
                                              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.

20.4 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.

20.5 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."

Part VII

In conversation

CHAPTER

TWENTYONE

FROM TREVOR, JOSIAH AND VÍCTOR

We are composers Trevor Bača, Josiah Wolf Oberholtzer 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

TWENTYTWO

DESIGNING ABJAD 2.14

22.1 Top-level functions, system protocol, parameterized tests

Trevor: How visible do you think the new code in 2.14 will be for composers already using Abjad?

Josiah: It's a good question. We've moved a lot of code around in 2.14. The `componenttools`, `leaftools`, `notetools`, `resttools`, `chordtools` and `tuplettools` packages have all been merged into `scoretools`. package.

Trevor: Which means `Note`, `Rest`, `Chord` and so on are all now housed in a single package.

Josiah: And we've replaced `iterationtools` with the new `iterate()` function.

Trevor: And `iotools` with `persist()`.

Josiah: Those changes will probably be more noticeable to existing users because most scores probably involve iteration. So users will have to change from thinking `iterationtools.iterate_notes_in_expr(voice)` to `iterate(voice).iterate_by_class(Note)`. Which is a cleaner way of getting to the core iteration functionality anyway.

Trevor: Same with `mutate()` and `inspect()`. Just those two functions offer up a tremendous amount of functionality.

Josiah: We should highlight `InspectionAgent`, `IterationAgent`, `MutationAgent` and `PersistenceAgent`. I feel pretty good about all of these changes, actually. I feel like this is a release of the system is really solid.

Trevor: We're almost there. We've removed something like 500 classes from public view in 2.13 and 2.14. And almost as many function. The learnability of the API is now dramatically higher than it's even been before.

Josiah: Even the typographic details of the code are better 2.14. I think we've implemented almost everywhere the pattern of putting everything on its own line and putting commas after every single expression. The The diffs are smaller and easier to read if you're watching the commits come into the repository during development.

Trevor: Also, the illustration protocol is really cool.

Josiah: Almost everything implements `__illustrate__()` now.

Trevor: Previously you could only `show(note)` and `show(tuplet)` and `show(score)`. But now you can `show(named_pitch)` and `show(markup)` and even `show(clef_inventory)`. It's powerful to be able to view so many parts of the system as music notation.

Josiah: I started adding those sorts of things into the `pitchtools` collection classes when I was cleaning up the classes for this release. (Classes like `pitchtools.Set` and `pitchtools.Segment`.) It just makes sense that you'd want to be able to see the pitches and pitch-classes that you're working with as you build up sets and segments.

Trevor: Just like it makes sense to be able to see notation corresponding to the clefs, range and name markup of all the different instruments. It took awhile to llustrate all the instrument classes. But the `instrumenttools` API is better because of it.

Josiah: We've got a number of different types of protocol in the system now.

Josiah: The illustration protocol. The copy protocol.

Trevor: The persistence protocol. The attachment protocol.

Josiah: The iteration protocol. The override protocol. The format protocol.

Trevor: The make-new protocol.

Josiah: The make-new protocol is a good one. We'll have to starting working the top-level `new()` function into examples. So that users can start to see it in action.

Trevor: All the functions in `topleveltools` deserve some attention: `attach()` and `detach()` to stick clefs and time signatures to the side of things; `show()` to show most things in the system; `mutate()` to make big changes to the score.

Josiah: `persist()` to write all sorts of objects to disk; `new()` to make new objects with optional changes.

Trevor: `graph()` to look at Graphviz pictures of things.

Josiah: The system has evolved to a point where a number of protocols are pretty important for how we do things. Essentially the entire formatting regime is protocol. I guess we didn't notice this upfront because the idea of a protocol is pretty unique to Python.

Trevor: I like that we've extended the Abjad initializer so that functions in `topleveltools` are all loaded into the global namespace now by default. You can just say `attach()`, `detach()`, `graph()`, `new()`, `persist()` and so on without having to import anything from anywhere.

Josiah: Same with `Articulation`, `Clef`, `KeySignature` and `TimeSignature`. It's really nice to be able to say `Clef('bass')` without having to worry about first importing `Clef` from somewhere.

Trevor: Right.

Josiah: The protocols are really helping write tests, too. I remember you put in an issue recently: Shouldn't we have a parameterized test that looks at everything that has a storage format and tries to make it write the storage format out, read the storage format back in and compare the equality of the two things?

Trevor: Exactly.

Josiah: Those tests exist now.

Trevor: Thanks to the `pytest` team giving us the `@pytest.mark.parameterize` decorator.

Josiah: Which is what makes all the new tests in the `tools/tests` directory work.

Trevor: Right.

Josiah: Have you given any thought to the new keyword-only arguments available in Python 3.0? Seems like many of the classes and functions now implemented in Abjad could benefit from being made keyword-only when we migrate to Python 3.0 later next year.

Trevor: I agree. Many of the more powerful classes in the system – the rhythm-makers, the quantizer, the instrument classes – allow for a number of input parameters. It makes sense to document the names of the input parameters everywhere. And the keyword-only arguments in Python 3.0 can help us enforce the pattern completely cleanly. It seems like a reasonable path would be to migrate all examples, all tests and all parts of the reference manual to keyword only sometime during the development of a major release in 2014. And then to enforce keyword-only arguments in the following major release.

Josiah: To show how the arguments will work in the documentation one release before requiring composers to start following the pattern.

[December 2013]

Part VIII

Appendices

CHAPTER

TWENTYTHREE

PITCH CONVENTIONS

23.1 Pitch numbers

Abjad numbers pitches like this:

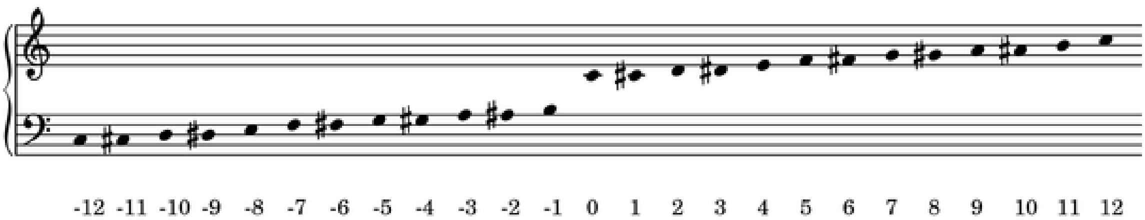
```
>>> score, treble_staff, bass_staff = scoretools.make_empty_piano_score()
>>> duration = Duration(1, 32)
```

```
>>> pitches = range(-12, 12 + 1)
>>> abjad_configuration.set_default_accidental_spelling('sharps')
```

```
>>> for pitch in pitches:
...     note = Note(pitch, duration)
...     rest = Rest(duration)
...     clef = pitchtools.suggest_clef_for_named_pitches([note.written_pitch])
...     if clef == indicatortools.Clef('treble'):
...         treble_staff.append(note)
...         bass_staff.append(rest)
...     else:
...         treble_staff.append(rest)
...         bass_staff.append(note)
...     pitch_number = note.written_pitch.pitch_number
...     markup = markuptools.Markup(str(pitch_number), Down)
...     attach(markup, bass_staff[-1])
...
```

```
>>> override(score).beam.transparent = True
>>> override(score).time_signature.stencil = False
>>> override(score).flag.transparent = True
>>> override(score).rest.transparent = True
>>> override(score).stem.stencil = False
>>> override(score).text_script.staff_padding = 6
>>> contextualize(score).proportional_notation_duration = schemetools.SchemeMoment(1, 56)
```

```
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
>>> lilypond_file.global_staff_size = 15
>>> show(lilypond_file)
```


-12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12

23.2 Diatonic pitch numbers

Abjad numbers diatonic pitches like this:

```
>>> score, treble_staff, bass_staff = scoretools.make_empty_piano_score()
>>> duration = Duration(1, 32)
```

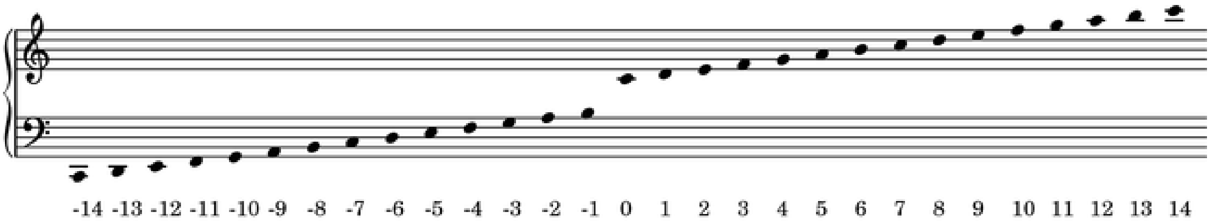
```
>>> pitches = []
>>> diatonic_pitches = [0, 2, 4, 5, 7, 9, 11]
```

```
>>> 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)
>>> abjad_configuration.set_default_accidental_spelling('sharps')
```

```
>>> for pitch in pitches:
...     note = Note(pitch, duration)
...     rest = Rest(duration)
...     clef = pitchtools.suggest_clef_for_named_pitches([note.written_pitch])
...     if clef == indicatortools.Clef('treble'):
...         treble_staff.append(note)
...         bass_staff.append(rest)
...     else:
...         treble_staff.append(rest)
...         bass_staff.append(note)
...     diatonic_pitch_number = note.written_pitch.diatonic_pitch_number
...     markup = markuptools.Markup(str(diatonic_pitch_number), Down)
...     attach(markup, bass_staff[-1])
...
```

```
>>> override(score).beam.transparent = True
>>> override(score).time_signature.stencil = False
>>> override(score).flag.transparent = True
>>> override(score).rest.transparent = True
>>> override(score).stem.stencil = False
>>> override(score).text_script.staff_padding = 6
>>> contextualize(score).proportional_notation_duration = schemetools.SchemeMoment(1, 52)
```

```
>>> lilypond_file = lilypondfiletools.make_basic_lilypond_file(score)
>>> lilypond_file.global_staff_size = 15
>>> show(lilypond_file)
```


-14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

23.3 Accidental abbreviations

Abjad abbreviates accidentals like this:

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’

23.4 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,,

23.5 Default accidental spelling

By default Abjad picks between enharmonic equivalents according to the following table:

pitch-class number	pitch-class name
0	C
1	C#
2	D
3	Eb
4	E
5	F
6	F#
7	G
8	Gb
9	A
10	Bb
11	B

You can change the default accidental spelling like this:

```
>>> abjad_configuration['default_accidental_spelling'] = 'sharps'
```

Or like this:

```
>>> abjad_configuration['default_accidental_spelling'] = 'sharps'
```

Or like this:

```
>>> abjad_configuration['default_accidental_spelling'] = 'mixed'
```

CHAPTER

TWENTYFOUR

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 Music. 1993.

[AndreattaBardezRahn2008] Moren Andreatta, Jean-Michel Bardez and John Rahn. *Around Set Theory.* Éditions Delatour, Paris. 2008.

[Ariza2005] Christopher Ariza. *An Open Design for Computer-Aided Algorithmic Music Composition: athenaCL.* Dissertation.com, Boca Raton. 2005.

[BačaAdan2007] 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.

[Forte1973] Alan Forte. The Structure of Atonal Music. Yale University Press, New Haven. 1973.

[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).

[MazzolaAhn2007] Guerino Mazzola and Yun-Kang Ahn. *La vérité du beau dans la musique.* Éditions Delatour, Paris. 2007.

[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.

[RebhahnSchäfer] Michael Rebhah and Thomas Schäfer. *Darmstädter Beiträge zur Neuen Musik.* vol 21. Schott Music GmbH & Co. KG, Mainz. 2012.

[RiotteMesnage2006-1] André Riotte and Marcel Mesnage. *Formalisme et modèles musicaux. 1* Éditions Delatour, Paris. 2006.

[RiotteMesnage2006-2] André Riotte and Marcel Mesnage. *Formalisme et modèles musicaux. 2* Éditions Delatour, Paris. 2006.

[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.

[Treviño2013] Jeffrey Treviño. *Compositional and Analytic Applications of Automated Music Notation via Object-oriented Programming.* Doctoral Dissertation. University of California San Diego. 2013.

[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."