Chapter II. MY COMPOSITIONAL PRACTICE WITH PYTHON, ABJAD, and LILYPOND

A. Methodology

In the preceding chapter, we have seen some of the strengths and potential weaknesses of Abjad and Lilypond when compared with similar programming paradigms, as well as some potential logical pitfalls when working with these programs. In my recent compositional practice, I have begun to amalgamate a workflow out of the ecosystem of Python, Abjad, and Lilypond, by learning from and embracing the idiosyncrasies of the software. The use of these tools in tandem is advantageous for my work due to the flexibility of Lilypond's notational algorithm and Abjad’s clarification of Lilypond’s model of music notation through Python’s object-oriented nature, as well as Python’s vast logical and mathematical abilities. Not only are Abjad and Lilypond both immensely rich in their features, but due to their open source nature, the source code for each is accessible to the user for further modification. Occasionally, I have found the need to tweak Abjad's source code in order for it to perform functions that I desire, but more often than this, the composer will find the need to build tools to simplify the process of engraving.

In my work, I often desire a structural rigor, where rhythms, pitches, and orchestration, among other parameters are balanced together by a plan or logic that gives meaning to potential musical realities. I am personally insecure when relying on the human system of intuition. A rigorous structure tends to fall apart when constructed by hand because humans are prone to err, while computers, conversely, do not make mistakes unless they are taught a false procedure. The computer does not have the ability to create a logical fallacy unless the error is programmed into its underlying functionality. Because of this, working with the Python programming language allows for a consistency in formal rigor that might be otherwise unattainable by intuition or hand-written calculations and graphs. It also allows for the potential modeling of complex systems and algorithmic music, where human intuition is placed in a more subordinate role to formal design.

Because of Lilypond’s ability to draw lines and shapes, and because it has a less restrictive model of notation than other software, it allows the composer to have greater graphic freedom. Another notable feature of Lilypond is its lack of a GUI, Graphic User Interface, allowing the program to spend more memory power when calculating spacing to avoid collisions, giving greater visual clarity upon the first engraving of a piece. Also, since it allows the user to include functions in the Scheme programming language, the user is able to affect other features like proportional spacing across an entire score instead of manually clicking and dragging note heads as one would do while using Finale or Sibelius. Lilypond has the ability to manage all visual aspects of a score and can also be used to export image files in the *pdf* and *png* formats, along with high quality *midi* files. Finally, a great feature of Lilypond is its context concatenation ability. As mentioned in the previous chapter, this allows multiple, separate Lilypond files to be combined with one another to stitch together separate segments of a full composition into one document.

An advantage to the Abjad composition paradigm is its ability to manage polyphony. Other programming paradigms like PatchWork Graphic Language or OpenMusic are a little more restricted in this regard. Often, in PWGL and OM, continuing a procedure from one instrument to the next is similar to the process of copying and pasting identical material to a different staff. To continue where one voice left off is a more difficult feat. This requires duplicating processes that were carried out in other voices, cluttering up the workspace with redundant information. In Abjad, the two concepts of copying and continuing are very distinct, allowing the composer to specifically use either technique as needed. Since Abjad is an API, or Application Programming Interface, in Python, it becomes very easy to cross-reference the same material-generating functions across different voices and at different points in time within the score. These strengths come from the fact that the music composed with Abjad is written as a text file. This allows the composer to create and manipulate any object or function they choose, whereas programs like PWGL and OM are slightly restricted by a GUI. Though there are ways for composers to write their own functions in these programs, it is more difficult to manipulate and it is not entirely obvious to a beginner that this is even possible. Because Abjad has no GUI, it inherently invites the composer to write the source code as part of the act of composition.

Though one could theoretically compose an entire score and only compile the Python file once the score is finalized, Abjad allows for an iterative workflow of composing, compiling, critiquing, and correcting in a cycle that lasts until the composer is satisfied with the composition. This workflow is reasonable because of the speed of modern computation as opposed to hand written calculation and engraving. One of the most attractive features of Abjad is that the system allows for the formalization of structures to control the placement and distribution of dynamics, articulations, and in fact, every visual element of the score. This is because Abjad attempts to model music notation rather than musical phenomenology. It treats all elements in a musical score as an object. An object in programming, as we have previously seen, has various attributes and potential modes of behavior. Some objects, like a note or a rest for instance, have a duration attribute, but a note has an attribute that a rest does not: pitch. Because all elements of the score are objects with properties and attributes, the entire score is manipulable via Abjad and, by extension, various formal means. This is a feature that is not present in OM and is difficult to achieve in PWGL, as OM does not display articulations or dynamics within the score viewing windows and PWGL’s interface is difficult to read. This is, in part, because these programs have different foci and goals. OM is typically used like a calculator for composers to generate options for materials with which to compose and PWGL, while able to export data to other notation engines, is equipped with its own ENP, Expressive Notation Package, with which music is rendered. Both OM and PWGL are based on CLOS, the Common LISP Object System, but I believe that the legibility of Python as well as the large number of Python programmers makes it a much better candidate for the user end of the system. Because the objects of notational elements are manipulable, they can be created, connected, and appended to one another throughout the composition process to create a score through composer-written procedures and functions as well as through built-in tools.

In this chapter, we will take a look at the compositional advantages of working with these programs such as how to automate potentially tedious tasks, the benefits of an iterative compositional workflow, and the possibilities for composing with algorithms or models. We will also look at some of my own solutions to composing with Abjad like my *MusicMaker* and *AttachmentHandler* classes as well as times when I have edited the Abjad source code. In the end, the greatest strength of this ecosystem is its flexibility.

1. How Is This Useful to Me as a Composer?

It is typical of my recent music to focus significantly on formal uniformity and continuous, alternating procedures. These procedures might be in relation to the rhythmic, harmonic, textural, or dynamic material. I have also become very interested in a pseudo-tablature style of notation that also features these iterative, procedural factors. Because these features are formalizable, it became apparent to me that I could leverage the programming concepts of loops and functions to write music very quickly. With this methodology, I have written various programs that organize and produce musical material based on my predetermined structures, allowing me to compose material and generate the product of these procedures in a very brief amount of time. In the course of my work in this manner, I have begun to appreciate the necessity of externalizing various tools in order to clean up my composition files. These tools, as well as my general compositional templates, could also easily be used by other composers, but they are tailored explicitly to my own compositional needs. Not only do my tools written in Python help me stay consistent with my formal designs, it also allows me to compose music that is specifically organized to my own tendencies and logic, rather than copying another composer’s tools and workflow. Although I have found a great amount of use out of the programs that I have written, they attempt to summarize behavioral activity with computational processes, thus these solutions are a work in progress and may not necessarily have universal functionality. All code examples in this paper are written in Python 3, Abjad 3.1, and Lilypond 2.19.82.

2. Automating Potentially Tedious Tasks

a. Creating Notes

An obvious first step in the creation of a score with Abjad is to ask the question: how does one make notes and then look at them? We have two options. We can open up the terminal, or command line, and activate a Python session so we can write our code or we could alternatively write our code in a text file saved with the *.py* suffix and call Python to compile it after we are done. The former method is better for quick testing of loops and materials, while the second method is much more sustainable for the process of composing a score, because it allows us to save our progress as well as multiple versions of our code along the way. Regardless of which method we choose, the code is written in the same way. The first step is always to import the Abjad API into our python session or file so that all of Abjad’s tools and properties are available to us. There are several ways of doing this, but the key to clarity is to be consistent. Throughout this chapter we will use this format:

>>> import abjad

This tells Python that we are going to be instantiating tools through the Abjad namespace. Doing this requires that we prefix all Abjad objects with *abjad.* followed by whatever object or tool we are using. Thus, a note object will look like this:

>>> abjad.Note()

We can give this note a variable name with which we are able to refer to the note throughout our file and we can use *abjad.show()* to quickly produce a *pdf* file of this note:

>>> import abjad

>>> note = abjad.Note()

>>> abjad.show(note)

This Abjad code will produce a Lilypond file containing the following text:

\version "2.19.82" %! LilyPondFile

\language "english" %! LilyPondFile

\header { %! LilyPondFile

tagline = ##f

} %! LilyPondFile

\layout {}

\paper {}

\score { %! LilyPondFile

{

c'4

}

} %! LilyPondFile

and will produce the following image in a *pdf* file:



As we can see, the note object has various default values associated with it. We are given a note with a pitch value of middle c and a duration value of one quarter note. Easily enough, these values are manipulable! We instead could have written:

>>> import abjad

>>> note = abjad.Note(11, abjad.Duration(1, 8))

>>> abjad.show(note)

from which we would receive the following Lilypond code:

\score { %! LilyPondFile

{

b'8

}

} %! LilyPondFile

and image:



So how then do we create many notes in a row in order to create a piece? First, we need to create a staff and notes. Then, we fill the staff with our notes and finally, show the staff. Here is one way we might do this:

>>> import abjad

>>> note\_1 = abjad.Note(0, abjad.Duration(1, 4))

>>> note\_2 = abjad.Note(1, abjad.Duration(1, 4))

>>> note\_3 = abjad.Note(2, abjad.Duration(1, 2))

>>> notes = [note\_1, note\_2, note\_3]

>>> staff = abjad.Staff(notes)

>>> abjad.show(staff)

from which we would receive the following Lilypond code:

\score { %! LilyPondFile

\new Staff

{

c'4

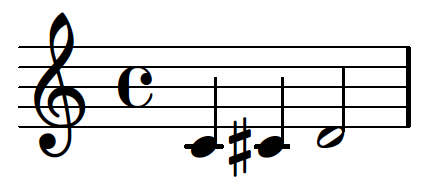
cs'4

d'2

}

} %! LilyPondFile

and image:



As you might begin to suspect, this process of note creation can get quite tedious. Here is one possible alternative approach to writing code with Abjad which is more economical for a longer piece, yet results in the same output in both Lilypond and the *pdf*:

>>> import abjad

>>> numerators = [1, 1, 1, ]

>>> denominators = [4, 4, 2, ]

>>> durations = [abjad.Duration(y, z) for y, z in zip(numerators, denominators)]

>>> pitches = [0, 1, 2, ]

>>> notes = [abjad.Note(x, y) for x, y in zip(pitches, durations)]

>>> note\_staff = abjad.Staff(notes)

>>> abjad.show(note\_staff)

Here we can see the use of *zip()* and the list comprehension we learned about in the first chapter. With *zip()* we create a list of numerators and denominators organized as tuples to represent fractions:

[(1, 4), (1, 4), (1, 2)]

and with the list comprehension we receive a list of duration objects based on those fractions:

[abjad.Duration((1, 4)), abjad.Duration((1, 4)), abjad.Duration((1, 2))]

we again zip together two lists, these being the list of pitches and the list of durations:

[(0, abjad.Duration((1, 4))), (1, abjad.Duration((1, 4))), (2, abjad.Duration((1, 2)))]

and create a note object for every pitch and duration in this list:

[abjad.Note(0, abjad.Duration((1, 4))), abjad.Note(1, abjad.Duration((1, 4))),

abjad.Note(2, abjad.Duration((1, 2)))]

we place this list of notes inside of a staff and show the staff. From this process, we receive the exact same Lilypond and image output:

\score { %! LilyPondFile

\new Staff

{

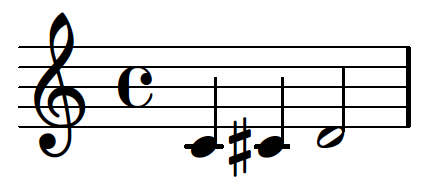
c'4

cs'4

d'2

}

} %! LilyPondFile



If we extrapolate this kind of process, we can begin to create loops to handle tasks of every shape and size! Because this process can be arduous at times, Abjad is equipped with a number of tools out of the box to assist in processes like note creation such as *abjad.LeafMaker(), abjad.NoteMaker(), abjad.MeasureMaker(),* and *abjad.SegmentMaker().* While these features are useful and are at the heart of many other tools like the Abjad-ext package *rmakers,* it is important to realize that it is not necessary to rely on these built-in functions to be able to write music with Abjad.

b. dynamics, articulations, and hairpins

Just like the creation of note objects, we can also simplify and formalize the attachment of dynamics:

>>> import abjad

>>> dynamic\_staff = abjad.Staff()

>>> dynamic\_staff.extend(r"c'4 cs'4 d'2")

>>> piano = abjad.Dynamic('p')

>>> mezzo\_forte = abjad.Dynamic('mf')

>>> forte = abjad.Dynamic('f')

>>> abjad.attach(piano, dynamic\_staff[0])

>>> abjad.attach(mezzo\_forte, dynamic\_staff[1])

>>> abjad.attach(forte, dynamic\_staff[2])

>>> abjad.show(dynamic\_staff)

resulting in the Lilypond code and image:

\score { %! LilyPondFile

\new Staff

{

c'4

\p

cs'4

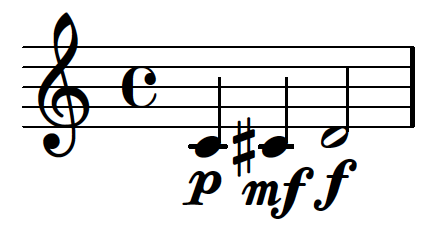
\mf

d'2

\f

}

} %! LilyPondFile



We can simplify this further by making use of a loop to attach the dynamics to each leaf in the staff, creating the dynamic object and attaching it at once:

>>> import abjad

>>> new\_staff = abjad.Staff()

>>> new\_staff.extend(r"c'4 cs'4 d'2")

>>> dynamics = ['p', 'mf', 'f', ]

>>> leaves = abjad.select(new\_staff).leaves()

>>> for leaf, dynamic in zip(leaves, dynamics):

...     abjad.attach(abjad.Dynamic(dynamic), leaf)

>>> abjad.show(new\_staff)

resulting in the Lilypond code and image:

\score { %! LilyPondFile

\new Staff

{

c'4

\p

cs'4

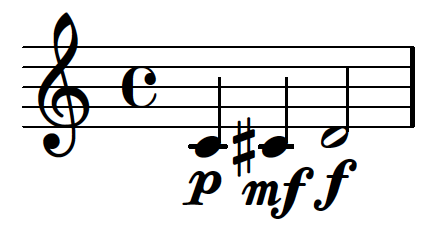
\mf

d'2

\f

}

} %! LilyPondFile



We have seen that dynamics behave in the same way as other attachable objects, but this is also true of articulations and hairpins. In the following example, we attach articulations and hairpins to our leaves as well, featuring a possible way to imbue some behavioral qualities into the attachment of these elements.

>>> import abjad

>>> music\_staff = abjad.Staff()

>>> music\_staff.extend(r"c'4 cs'4 d'2 r4 ds'2. e'8 f'8 fs'8 g'8 gs'8 r4. a'1")

>>> for run in abjad.select(music\_staff).runs():

... if len(run) > 3:

... leaves = abjad.select(run).leaves()

... abjad.attach(abjad.Dynamic('mf'), run[0])

... for leaf in leaves:

... abjad.attach(abjad.Articulation('tenuto'), leaf)

... elif len(run) == 3:

... abjad.attach(abjad.Dynamic('f'), run[0])

... abjad.attach(abjad.StartHairpin('>'), run[0])

... abjad.attach(abjad.Dynamic('mp'), run[-1])

... elif len(run) == 1:

... abjad.attach(abjad.Dynamic('ppp'), run[0])

>>> abjad.show(music\_staff)

resulting in the Lilypond code and image:

\score { %! LilyPondFile

\new Staff

{

c'4

\f

\>

cs'4

d'2

\mp

r4

e'2

\mf

- \tenuto

f'8

- \tenuto

g'8

- \tenuto

a''8

- \tenuto

b''8

- \tenuto

c''8

- \tenuto

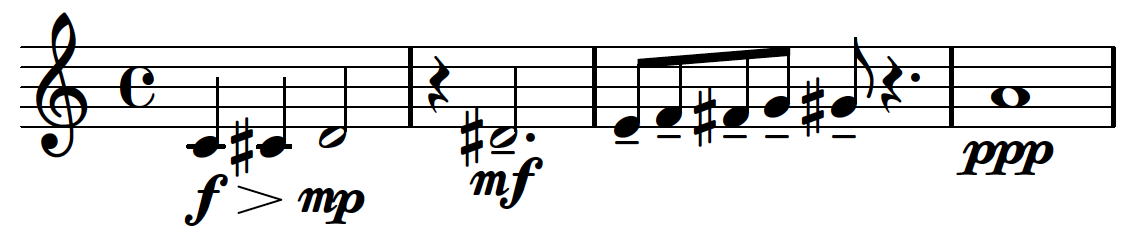
r4

c''2.

\ppp

}

} %! LilyPondFile



This loop analyzes the length of each run in the staff and chooses what dynamics and articulations to attach based on the result. This is an extremely powerful method for attaching indicators throughout a score. Next, I will address howwe might write a procedure to handle the *abjad.BowContactPoint()* object, which produces a more complex Lilypond result and graphic.

c. Using *abjad.BowContactPoint()*

The *abjad.BowContactPoint()* object and an accompanying factory class, *abjad.bow\_contact\_spanner(),*  are tools that are able to annotate a staff of notes with fractions intended to represent points along the length of a bow. Native in these tools is the ability to calculate whether one fraction is greater or lesser than its surrounding fractions and attach an upbow or downbow marking as needed. Because of this feature, I created a file in Abjad 2.21 which I called *abjad.StringContactSpanner* which eliminated the bow markings in order for it to be used universally for any potential parameter. This file was adapted by Trevor Bača into Abjad 3.1’s *abjad.BowContactPoint* which features an optional keyword to include or exclude these bowings. Here is a possible way to use these tools:

>>> import abjad

>>> bow\_staff = abjad.Staff()

>>> bow\_staff.extend(r"c'4 c'4 c'4 c'4")

>>> indicator\_1 = abjad.BowContactPoint((3, 3))

>>> indicator\_2 = abjad.BowContactPoint((2, 3))

>>> indicator\_3 = abjad.BowContactPoint((1, 3))

>>> indicator\_4 = abjad.BowContactPoint((0, 3))

>>> abjad.attach(indicator\_1, bow\_staff[0])

>>> abjad.attach(indicator\_2, bow\_staff[1])

>>> abjad.attach(indicator\_3, bow\_staff[2])

>>> abjad.attach(indicator\_4, bow\_staff[3])

>>> abjad.bow\_contact\_spanner(bow\_staff, omit\_bow\_changes=True)

>>> abjad.show(bow\_staff)

resulting in the Lilypond code:

\score { %! LilyPondFile

\new Staff

{

\tweak Y-offset #2.0

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

1

1

}

c'4

\glissando

\tweak Y-offset #0.6666666666666666

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

2

3

}

c'4

\glissando

\tweak Y-offset #-0.6666666666666666

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

1

3

}

c'4

\glissando

\tweak Y-offset #-2.0

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

0

1

}

c'4

}

} %! LilyPondFile

and image:



We can see several, lengthy *\tweak* commands in the Lilypond code. Imagine composing a score in Lilypond where an instrument has two staves, one of which is a bowing tablature that uses notation similar to what is produced by the *abjad.BowContactPoint()* tool. This Lilypond code could easily get even more tedious to write than the note creation process above, making this tool quite useful for speeding up the engraving process. The following examples are a few alternative methods that achieve the same results in a similar manner of reduction as in the note creation examples:

>>> import abjad

>>> new\_bow\_staff = abjad.Staff()

>>> new\_bow\_staff.extend(r"c'4 c'4 c'4 c'4")

>>> indicator\_1 = abjad.BowContactPoint((3, 3))

>>> indicator\_2 = abjad.BowContactPoint((2, 3))

>>> indicator\_3 = abjad.BowContactPoint((1, 3))

>>> indicator\_4 = abjad.BowContactPoint((0, 3))

>>> indicators = [indicator\_1, indicator\_2, indicator\_3, indicator\_4, ]

>>> leaves = abjad.select(new\_bow\_staff).leaves()

>>> for leaf, indicator in zip(leaves, indicators):

... abjad.attach(indicator, leaf)

>>> abjad.bow\_contact\_spanner(new\_bow\_staff, omit\_bow\_changes=True)

>>> abjad.show(new\_bow\_staff)

resulting in the Lilypond code:

\score { %! LilyPondFile

\new Staff

{

\tweak Y-offset #2.0

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

1

1

}

c'4

\glissando

\tweak Y-offset #0.6666666666666666

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

2

3

}

c'4

\glissando

\tweak Y-offset #-0.6666666666666666

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

1

3

}

c'4

\glissando

\tweak Y-offset #-2.0

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

0

1

}

c'4

}

} %! LilyPondFile

and image:



This example is very similar to the previous example, except for the fact that the process of attaching indicators to leaves has been streamlined. Here is another possibility:

>>> import abjad

>>> new\_bow\_staff = abjad.Staff()

>>> new\_bow\_staff.extend(r"c'4 c'4 c'4 c'4")

>>> numerators = [3, 2, 1, 0, ]

>>> indicators = [(abjad.BowContactPoint((numerator, 3))) for numerator in numerators]

>>> leaves = abjad.select(new\_bow\_staff).leaves()

>>> for leaf, indicator in zip(leaves, indicators):

... abjad.attach(indicator, leaf)

>>> abjad.bow\_contact\_spanner(new\_bow\_staff, omit\_bow\_changes=True)

>>> abjad.show(new\_bow\_staff)

resulting in the Lilypond code and image:

\score { %! LilyPondFile

\new Staff

{

\tweak Y-offset #2.0

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

1

1

}

c'4

\glissando

\tweak Y-offset #0.6666666666666666

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

2

3

}

c'4

\glissando

\tweak Y-offset #-0.6666666666666666

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

1

3

}

c'4

\glissando

\tweak Y-offset #-2.0

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

0

1

}

c'4

}

} %! LilyPondFile



Here we see a further simplification. In this code, the fractions in the indicators are summarized in a list comprehension. If we simplify this process even further we can write code like this:

>>> import abjad

>>> newer\_bow\_staff.extend(r"c'4 c'4 c'4 c'4")

>>> leaves = abjad.select(newer\_bow\_staff).leaves()

>>> indicator\_numerators = [3, 2, 1, 0, ]

>>> for leaf, numerator in zip(leaves, indicator\_numerators):

... abjad.attach(abjad.BowContactPoint((numerator, 3)), leaf)

>>> abjad.bow\_contact\_spanner(newer\_bow\_staff, omit\_bow\_changes=True)

>>> abjad.show(newer\_bow\_staff)

resulting in the Lilypond code and image:

\score { %! LilyPondFile

\new Staff

{

\tweak Y-offset #2.0

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

1

1

}

c'4

\glissando

\tweak Y-offset #0.6666666666666666

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

2

3

}

c'4

\glissando

\tweak Y-offset #-0.6666666666666666

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

1

3

}

c'4

\glissando

\tweak Y-offset #-2.0

\tweak stencil #ly:text-interface::print

\tweak text \markup {

\center-align

\vcenter

\fraction

0

1

}

c'4

}

} %! LilyPondFile



This version of our code is comprised of the least number of lines. We summarize both the fractions and the attachment processes in a loop that takes our fractions, puts them in an indicator and attaches those indicators each to a leaf of the staff. Notice that each version of our code results in the same output, but each option simplifies the process. Extensive use of *abjad.BowContactPoint()* can be found in the compositions *Armilla* by Josiah Wolf Oberholtzer and *Cthar* by myself.

d. stylesheets

An important concept when working with Lilypond is the idea of a stylesheet. Typically, the beginning of each Lilypond file will be full of information telling Lilypond how to format and render the music within the file. To make use of Lilypond’s context concatenation ability, it is best to externalize this information into a file called a stylesheet. We use an *\include* statement to let Lilypond know where to find this information. The stylesheet is written in Lilypond syntax and occasionally Scheme code and may feature information about horizontal spacing proportional to the duration of notes, vertical spacing in staff groups, the removal of time signatures within staves, and the creation of a new context for displaying those time signatures above the staff group. This is also where information about font, font size, paper size and orientation, and header information is stored. The following is the stylesheet that I wrote for my cello duo *Cthar*:

% 2018-07-17 19:54

\version "2.19.82"

\language "english"

#(set-default-paper-size "letterlandscape")

#(set-global-staff-size 10)

\include "ekmel.ily"

\ekmelicStyle evans

\header {

tagline = ##f

breakbefore = ##t

title = \markup \override #'(font-name . "Didot") \fontsize #15 \bold \center-column {"Cthar"}

subtitle = \markup \override #'(font-name . "Didot") \fontsize #4 \center-column {"for two cellos"}

arranger = \markup \override #'(font-name . "Didot") \fontsize #2.5 {"Gregory Rowland Evans"}

}

bowtab = {

\override Staff.Clef.stencil = #ly:text-interface::print

\override Staff.Clef.text = \markup { \general-align #Y #0.03

\epsfile #Y #10 #"bow\_position\_tablature.eps"

}

}

\layout {

\accidentalStyle forget

indent = #5

ragged-right = ##t

\context {

\name TimeSignatureContext

\type Engraver\_group

\numericTimeSignature

\consists Axis\_group\_engraver

\consists Bar\_number\_engraver

\consists Time\_signature\_engraver

\consists Mark\_engraver

\consists Metronome\_mark\_engraver

\override BarNumber.Y-extent = #'(0 . 0)

\override BarNumber.Y-offset = 0

\override BarNumber.extra-offset = #'(-4 . 0)

%\override BarNumber.font-name = "Didot"

\override BarNumber.stencil = #(make-stencil-boxer 0.1 0.7 ly:text-interface::print)

\override BarNumber.font-size = 1

\override BarNumber.padding = 4

\override MetronomeMark.X-extent = #'(0 . 0)

\override MetronomeMark.Y-extent = #'(0 . 0)

\override MetronomeMark.break-align-symbols = #'(left-edge)

\override MetronomeMark.extra-offset = #'(0 . 4)

\override MetronomeMark.font-size = 10

\override RehearsalMark.stencil = #(make-stencil-circler 0.1 0.7 ly:text-interface::print)

\override RehearsalMark.X-extent = #'(0 . 0)

\override RehearsalMark.X-offset = 6

\override RehearsalMark.Y-offset = -2.25

\override RehearsalMark.break-align-symbols = #'(time-signature)

\override RehearsalMark.break-visibility = #end-of-line-invisible

\override RehearsalMark.font-name = "Didot"

\override RehearsalMark.font-size = 8

\override RehearsalMark.outside-staff-priority = 500

\override RehearsalMark.self-alignment-X = #center

\override TimeSignature.X-extent = #'(0 . 0)

\override TimeSignature.X-offset = #ly:self-alignment-interface::x-aligned-on-self

\override TimeSignature.Y-extent = #'(0 . 0)

\override TimeSignature.Y-offset = 3

\override TimeSignature.break-align-symbol = ##f

\override TimeSignature.break-visibility = #end-of-line-invisible

\override TimeSignature.font-size = #7

\override TimeSignature.self-alignment-X = #center

\override VerticalAxisGroup.default-staff-staff-spacing = #'((basic-distance . 0) (minimum-distance . 10) (padding . 6) (stretchability . 0))

}

\context {

\Score

\remove Bar\_number\_engraver

\remove Mark\_engraver

\accepts TimeSignatureContext

\accepts LipStaff

\override BarLine.bar-extent = #'(-2 . 2)

\override Beam.breakable = ##t

\override Beam.concaveness = #10000

\override Glissando.breakable = ##t

\override MetronomeMark.font-size = 5

\override SpacingSpanner.strict-grace-spacing = ##t

\override SpacingSpanner.strict-note-spacing = ##t

\override SpacingSpanner.uniform-stretching = ##t

\override StaffGrouper.staff-staff-spacing = #'((basic-distance . 0) (minimum-distance . 6) (padding . 2))

\override TupletBracket.bracket-visibility = ##t

\override TupletBracket.minimum-length = #3

\override TupletBracket.padding = #2

\override TupletBracket.springs-and-rods = #ly:spanner::set-spacing-rods

\override TupletNumber.text = #tuplet-number::calc-fraction-text

\override TextSpanner.Y-offset = 1

proportionalNotationDuration = #(ly:make-moment 1 50)

autoBeaming = ##f

tupletFullLength = ##t

}

\context {

\Voice

\remove Forbid\_line\_break\_engraver

}

\context {

\Staff

\remove Time\_signature\_engraver

}

\context {

\Staff

\name BowStaff

\type Engraver\_group

\alias Staff

\bowtab

\override Beam.stencil = ##f

\override Dots.stencil = ##f

\override Flag.stencil = ##f

\override Glissando.bound-details.left.padding = #0.5

\override Glissando.bound-details.right.padding = #0.5

\override Glissando.thickness = #2

\override NoteHead.Y-offset = #-5

\override NoteHead.extra-offset = #'(0.05 . 0)

\override NoteHead.stencil = ##f

\override Rest.transparent = ##t

\override Script.staff-padding = #2

\override StaffSymbol.transparent = ##t

\override Stem.direction = #down

\override Stem.stencil = ##f

\override TimeSignature.stencil = ##f

\override Tie.stencil = ##f

\override TupletBracket.stencil = ##f

\override TupletNumber.stencil = ##f

%\RemoveEmptyStaves

}

\context {

\Staff

\name BeamStaff

\type Engraver\_group

\alias Staff

\override Beam.direction = #down

\override Beam.positions = #'(5 . 5)

\override Clef.stencil = ##f

\override Dots.staff-position = #-2

\override Flag.Y-offset = #2.93

\override NoteHead.no-ledgers = ##t

\override NoteHead.stencil = ##f

\override Rest.transparent = ##t

\override Script.staff-padding = #3

\override StaffSymbol.transparent = ##t

\override Stem.direction = #down

\override Stem.length = #0.5

\override Stem.stem-begin-position = #15.975

\override TimeSignature.stencil = ##f

\override Tie.stencil = ##f

\override TupletBracket.positions = #'(3 . 3)

}

\context {

\RhythmicStaff

\remove Time\_signature\_engraver

}

\context {

\StaffGroup

\accepts BowStaff

\accepts BeamStaff

}

}

\paper {

top-margin = 1.5\cm

bottom-margin = 1.5\cm

%top-margin = .90\in

oddHeaderMarkup = \markup ""

evenHeaderMarkup = \markup ""

oddFooterMarkup = \markup \fill-line {

""

\concat {

"Cthar ~"

\fontsize #2

\fromproperty #'page:page-number-string "~ Evans"

}

""

}

evenFooterMarkup = \markup \fill-line {

""

\concat { "Cthar ~" \fontsize #2

\fromproperty #'page:page-number-string "~ Evans"

} ""

}

}

In this score, I defined a few new contexts in order to manage the specific visual properties I desired for a staff indicating bow motion with the *abjad.BowContactPoint()* tool. Aside from these properties, the composer is also able to edit graphic elements such as the width and spacing of beams, the thickness of stems, or the shape of flags.

These are just a few examples of ways in which Abjad and Lilypond allow for the simplification of processes that, by hand, could be extremely tedious over the course of a lengthy composition. The principles involved in these examples extend to every facet of both composing and engraving. Now that we have seen the power that Python can give composers, next we will see how creating these loops and functions has further ramifications in the process of composing.

3. Composing with Algorithms and Models

Composing with Abjad and Python allows the composer to work with algorithms and models. Next is an example where pitches are generated by a random walk which can be seen as a one-dimensional model of Brownian Motion. Much of my recent music features a similar procedure as the following:

>>> import abjad

>>> from random import seed

>>> from random import random

>>> seed(3)

>>> random\_walk = []

>>> random\_walk.append(-1 if random() < 0.5 else 1)

>>> for i in range(1, 64):

... movement = -1 if random() < 0.5 else 1

... value = random\_walk[i-1] + movement

... random\_walk.append(value)

>>> notes = [abjad.Note(x / 2.0, (1, 8)) for x in random\_walk]

>>> staff = abjad.Staff(notes)

>>> abjad.show(staff)

In this code, we create an empty list. Based on a string of randomly generated numbers, we create a new list of pitches notated by numbers moving in a step of plus or minus 0.5 that are turned into note objects that are placed in a staff. When the staff is shown, it results in this Lilypond code and image:

\score { %! LilyPondFile

\new Staff

{

bqs8

c'8

bqs8

c'8

cqs'8

c'8

bqs8

c'8

bqs8

b8

bqs8

b8

bqs8

b8

bqs8

b8

bqs8

c'8

cqs'8

cs'8

dqf'8

cs'8

dqf'8

d'8

dqf'8

cs'8

dqf'8

cs'8

dqf'8

d'8

dqs'8

ef'8

dqs'8

ef'8

dqs'8

ef'8

eqf'8

ef'8

dqs'8

d'8

dqs'8

d'8

dqs'8

d'8

dqs'8

d'8

dqf'8

d'8

dqs'8

ef'8

eqf'8

e'8

eqs'8

f'8

fqs'8

f'8

fqs'8

fs'8

gqf'8

g'8

gqs'8

g'8

gqs'8

af'8

}

} %! LilyPondFile



We can also model more traditional compositional algorithms. The following code is adapted from code written by Jeffrey Treviño presented as a part of the 2018 Abjad summer workshop at CCRMA at Stanford University. This code is more complex than what we have seen before. This code creates a three-voice canon based on the melody input by the user. The melody is transposed and the rhythms are scaled to a different tempo. Voices with phrases that end before the slowest voice completes its phrase are repeated until the bottom voice has finished. Because of how the rhythms are scaled, we need to use *abjad.mutate().rewrite\_meter()* to ensure that all rhythms remain in the appropriate measure:

>>> import abjad

>>> def generate\_scaled\_staff(scale\_factor, staff):

... staff\_pitches = []

... for logical\_tie in abjad.iterate(staff).logical\_ties():

... first\_leaf = logical\_tie[0]

... staff\_pitches.append(first\_leaf.written\_pitch)

... staff\_durations = [chain.written\_duration\*scale\_factor for chain in abjad.iterate(staff).logical\_ties()]

... scaled\_staff = abjad.Staff()

... maker = abjad.NoteMaker()

... selections = maker(staff\_pitches, staff\_durations)

... scaled\_staff.extend(selections)

... return scaled\_staff

>>> def partition\_value(value):

... if x >= 16:

... divisions, remainder = divmod(value, 8)

... parts = [8] \* divisions

... if remainder:

... parts.append(remainder)

... return parts

>>> def process\_maxima(durations):

... output\_durations = []

... for duration in durations:

... if duration[0] >= 16:

... numerators = partition\_value(duration[0])

... duration = [(numerator, 1) for numerator in numerators]

... output\_durations.append(duration)

>>> def scale\_and\_chop\_staff(voice\_number, staff, time\_signature):

... scale\_factor = 2 \*\* voice\_number

... scaled\_staff = generate\_scaled\_staff(scale\_factor, staff)

... abjad.mutate(scaled\_staff).transpose(voice\_number \* -7)

... abjad.mutate(scaled\_staff[:]).split([time\_signature], cyclic=True)

... return scaled\_staff

>>> def duplicate\_music(num\_copies, staff):

... out\_staff = abjad.Staff()

... for x in range(num\_copies):

... out\_staff.extend(abjad.mutate(staff).copy())

... return out\_staff

>>> def make\_scaled\_staves(melody\_staff, time\_signature):

... scaled\_staves = []

... for voice\_number in range(3):

... scaled\_staff = scale\_and\_chop\_staff(voice\_number, melody\_staff, time\_signature)

... scaled\_staves.append(scaled\_staff)

... return scaled\_staves

>>> def duplicate\_score(scaled\_staves):

... score = abjad.Score()

... for scaled\_staff, duplicate\_index in zip(scaled\_staves, reversed(range(3))):

... scale\_factor = 2\*\*duplicate\_index

... staff = duplicate\_music(scale\_factor, scaled\_staff)

... score.append(staff)

... return score

>>> def format\_score(score, key\_signature, time\_signature):

... for staff in score:

... key\_sig = abjad.KeySignature(key\_signature.tonic, key\_signature.mode)

... abjad.attach(key\_sig, staff[0])

... time\_sig = abjad.TimeSignature(time\_signature)

... abjad.attach(time\_sig, staff[0])

... abjad.attach(abjad.Clef('varC'), score[1][0])

... abjad.attach(abjad.Clef('bass'), score[2][0])

>>> def make\_canon(melody\_staff, key\_signature, time\_signature):

... scaled\_staves = make\_scaled\_staves(melody\_staff, time\_signature)

... score = duplicate\_score(scaled\_staves)

... format\_score(score, key\_signature, time\_signature)

... return score

>>> def rewrite\_meter(score):

... meter = abjad.Meter()

... for staff in score:

... for shard in abjad.mutate(staff[:]).split([abjad.Duration(4, 4)], cyclic=True):

... abjad.mutate(shard).rewrite\_meter(meter)

>>> melody\_staff = abjad.Staff("c'4 cs'8 d' ds' e' f'4 fs' g' gs'8 a' b' c''")

>>> score = make\_canon(melody\_staff, abjad.KeySignature('c', 'major'), abjad.TimeSignature((4,4)))

>>> rewrite\_meter(score)

>>> abjad.show(score)

resulting in the Lilypond code and image:

\score { %! LilyPondFile

\new Score

<<

\new Staff

{

\key c \major

\time 4/4

c'4

cs'8

d'8

ds'8

e'8

f'4

fs'4

g'4

gs'8

a'8

b'8

c''8

c'4

cs'8

d'8

ds'8

e'8

f'4

fs'4

g'4

gs'8

a'8

b'8

c''8

c'4

cs'8

d'8

ds'8

e'8

f'4

fs'4

g'4

gs'8

a'8

b'8

c''8

c'4

cs'8

d'8

ds'8

e'8

f'4

fs'4

g'4

gs'8

a'8

b'8

c''8

}

\new Staff

{

\key c \major

\time 4/4

\clef "varC"

f2

fs4

g4

gs4

a4

bf2

b2

c'2

cs'4

d'4

e'4

f'4

f2

fs4

g4

gs4

a4

bf2

b2

c'2

cs'4

d'4

e'4

f'4

}

\new Staff

{

\key c \major

\time 4/4

\clef "bass"

bf,1

b,2

c2

cs2

d2

ef1

e1

f1

fs2

g2

a2

bf2

}

>>

} %! LilyPondFile



Using Abjad and Python, we are able to compose music full of intricate relationships with extreme formal consistency, but a comfortable formalism in score control is not necessarily algorithmic utopia. Though these logical procedures are available and entirely possible, they are optional. Writing a loop to create a list of note objects hardly qualifies as being the foundation of an algorithmic composition. This process should not be misperceived as a purely algorithmic system for music composition. Certainly, formalizing elements in a score allows for a great amount of consistency and control, but the composer has every ability to make decisions and sculpt the music at will if they so desire. Composing with the workflow of Python, Abjad, and Lilypond does present some difficulty in composing idiomatically for instruments. Piano music, in particular, presents a great challenge, a challenge that I have yet to surmount. If one is not careful, it is possible to compose music completely unplayable by a human performer. Abjad and Lilypond do not dictate what kind of music is able to be composed. It is still the duty of the composer to constrain their musical practices to those they consciously wish to deploy.

B. The need to build tools for a more personalized approach to music-making

1. Why should composers build their own tools?

Why is it important for composers to write their own compositional tools? This is because each composer has a unique imagination and ideal. If every composer was expected to compose with the same methodology, no composer would have a unique voice and the beautiful diversity of new music would vanish. Abjad provides a framework for formalized score control but is not restrictive about the practices used to compose. Even composing in notation engines such as Finale or Sibelius has restrictions and makes certain procedures difficult or impossible. The fact that Abjad provides separate packages of tools for composition, as well as other functions, reveals that it is intended to be used by a variety of users with a variety of backgrounds. There are a handful of official extensions to Abjad under the title of Abjad-ext.

2. abjad-ext

a. What is abjad-ext?

Abjad-ext consists of a number of packages that are not necessary for full functionality of the API. The packages include *abjad-ext-tonality*, a tonal analysis extension, *abjad-ext-book*, an extension for rendering Abjad code in LaTeX, *abjad-ext-ipython*, an extension for rendering Abjad code in IPhython and Jupyter notebooks, *abjad-ext-nauert*, and extension of quantization tools based on Paul Nauert’s Q-Grids, *abjad-ext-cli*, a Command Line Interface extension, and *abjad-ext-rmakers*, a rhythm maker tool extension. Each of these packages extend the functionality of Abjad, but I have only seriously used Trevor Bača’s *rmakers* package. These packages exist outside of the main Abjad source in order to emphasize their optionality. The *rmakers*, previously called *RhythmMakerTools*, were once a part of the main Abjad source, but were externalized because Trevor Bača felt they were more of a reflection of his own compositional practices than being a universal tool.

b. rmakers

The *rmakers* consist of a set of tools for generating rhythmic material in certain characteristic ways. The *rmakers* themselves are a basic *RhythmMaker* class, *AccelerandoRhythmMaker, EvenDivisionRhythmMaker, IncisedRhythmMaker, NoteRhythmMaker, TaleaRhythmMaker, and TupletRhythmMaker.* An extended description of these tools and their functionality can be found in Josiah Wolf Oberholtzer’s 2015 dissertation *A Computational Model of Music Composition*. I am quite fond of these tools and, even though I intend to write my own rhythm-generating functions in the near future, they are the primary source of rhythmic composition in my recent music. Along with the official Abjad-ext packages are other packages by composers who make use of Abjad, including *Consort* a package written by Josiah Wolf Oberholtzer and described in detail in his dissertation, and *mtools* by Ivan Alexander Moscotta. All of these packages present unique and innovative tools for music composition and have encouraged me to find my own way of composing with Abjad. As of the writing of this paper, I have only written two external tools for composing, but they are used extensively in my scores, these are *MusicMaker* and *AttachmentHandler.*

3. MusicMaker

MusicMaker is my attempt to combine gestural consistency of many kinds. MusicMaker takes the input of an rmaker and an optional AttachmentHandler. I wrote this tool because, although the rmakers are capable of generating rhythmic material, they do not handle pitch in any way. One could compose the entire rhythmic framework of a piece and add pitches after the fact, but I found myself using many different rmakers throughout the course of a piece. I also found that I was working very hard to unify rhythmic gestures by giving them unique harmonic fields and dynamic trajectories. When MusicMaker is given an rmaker and a list of pitches, it automatically adds those pitches to the rhythms. Whenever this MusicMaker is called in the Python file, it generates music based on the rhythms and pitches that were input by the user, allowing the composer to instantiate multiple MusicMakers with unique rhythmic, harmonic, and dynamic qualities. Much of my recent music concerns alternating between fragments of processes begun with MusicMaker. As I write more music, I find that there are more features that I wish to add to MusicMaker and AttachmentHandler. As such, their code is still under revision. The following is the current source code for MusicMaker:

import abjad

from AttachmentHandler import AttachmentHandler

class MusicMaker:

def \_\_init\_\_(

self,

rmaker,

attachment\_handler=None,

pitches=None,

continuous=False,

state=None,

):

self.attachment\_handler = attachment\_handler

self.rmaker = rmaker

self.pitches = pitches

self.continuous = continuous

self.state = self.rmaker.state

self.\_count = 0

def \_\_call\_\_(self, durations):

return self.\_make\_music(durations)

def \_make\_basic\_rhythm(self, durations):

state = self.state

selections = self.rmaker(durations, previous\_state=self.rmaker.state)

self.state = self.rmaker.state

return selections

def \_make\_music(self, durations):

selections = self.\_make\_basic\_rhythm(durations)

if self.pitches == None:

start\_command = abjad.LilyPondLiteral(

r'\stopStaff \once \override Staff.StaffSymbol.line-count = #0 \startStaff',

format\_slot='before',

)

stop\_command = abjad.LilyPondLiteral(

r'\stopStaff \startStaff',

format\_slot='after',

)

abjad.attach(start\_command, selections[0][0])

abjad.attach(stop\_command, selections[0][-1])

if self.pitches != None:

selections = self.\_apply\_pitches(selections, self.pitches)

if self.attachment\_handler != None:

selections = self.attachment\_handler(selections)

self.\_count += 1

return selections

def \_collect\_pitches\_durations\_leaves(self, logical\_ties, pitches):

def cyc(lst):

if self.continuous == False:

self.\_count = 0

while True:

yield lst[self.\_count % len(lst)]

self.\_count += 1

cyc\_pitches = cyc(pitches)

pitches, durations, leaves = [[], [], []]

for tie in logical\_ties:

if isinstance(tie[0], abjad.Note):

pitch = next(cyc\_pitches)

for leaf in tie:

pitches.append(pitch)

durations.append(leaf.written\_duration)

leaves.append(leaf)

else:

for leaf in tie:

pitches.append(None)

durations.append(leaf.written\_duration)

leaves.append(leaf)

return pitches, durations, leaves

def \_apply\_pitches(self, selections, pitches):

leaf\_maker = abjad.LeafMaker()

container = abjad.Container(selections)

old\_ties = [tie for tie in abjad.iterate(

container).logical\_ties()]

pitches, durations, old\_leaves = self.\_collect\_pitches\_durations\_leaves(

old\_ties, pitches)

new\_leaves = [leaf for leaf in leaf\_maker(pitches, durations)]

for old\_leaf, new\_leaf in zip(old\_leaves, new\_leaves):

indicators = abjad.inspect(old\_leaf).indicators()

for indicator in indicators:

abjad.attach(indicator, new\_leaf)

parent = abjad.inspect(old\_leaf).parentage().parent

parent[parent.index(old\_leaf)] = new\_leaf

return [container[:]]

MusicMaker can be used with timespans as follows:

>>> import abjad

>>> import itertools

>>> import abjadext.rmakers

>>> from MusicMaker import MusicMaker

>>> from AttachmentHandler import AttachmentHandler

>>> time\_signatures = [

... abjad.TimeSignature(pair) for pair in [

... (4, 4), (5, 4),

... ]

...]

>>> bounds = abjad.mathtools.cumulative\_sums([\_.duration for \_ in time\_signatures])

>>> rmaker\_one = abjadext.rmakers.TaleaRhythmMaker(

... talea=abjadext.rmakers.Talea(

... counts=[1, 2, 3, 4],

... denominator=16,

... ),

... beam\_specifier=abjadext.rmakers.BeamSpecifier(

... beam\_divisions\_together=True,

... beam\_rests=False,

... ),

... extra\_counts\_per\_division=[0, 1,],

... tuplet\_specifier=abjadext.rmakers.TupletSpecifier(

... trivialize=True,

... extract\_trivial=True,

... rewrite\_rest\_filled=True,

... ),

... )

>>> rmaker\_two = abjadext.rmakers.EvenDivisionRhythmMaker(

... denominators=[8, 16,],

... extra\_counts\_per\_division=[0,],

... tuplet\_specifier=abjadext.rmakers.TupletSpecifier(

... trivialize=True,

... extract\_trivial=True,

... rewrite\_rest\_filled=True,

... ),

... )

>>> musicmaker\_one = MusicMaker(

... rmaker=rmaker\_one,

... pitches=[0, 1, 2, 3, 4],

... continuous=True,

...)

>>> musicmaker\_two = MusicMaker(

... rmaker=rmaker\_two,

... pitches=[4, 3, 2, 1, 0],

... continuous=True,

...)

>>> silence\_maker = abjadext.rmakers.NoteRhythmMaker(

... division\_masks=[

... abjadext.rmakers.SilenceMask(

... pattern=abjad.index([0], 1),

... ),

... ],

... )

>>> class MusicSpecifier:

... def \_\_init\_\_(self, music\_maker, voice\_name):

... self.music\_maker = music\_maker

... self.voice\_name = voice\_name

>>> voice\_1\_timespan\_list = abjad.TimespanList([

... abjad.AnnotatedTimespan(

... start\_offset=start\_offset,

... stop\_offset=stop\_offset,

... annotation=MusicSpecifier(

... music\_maker=music\_maker,

... voice\_name='Voice 1',

... ),

... )

>>> for start\_offset, stop\_offset, music\_maker in [

... [(0, 4), (2, 4), musicmaker\_one],

... [(2, 4), (3, 4), musicmaker\_one],

... [(3, 4), (4, 4), musicmaker\_one],

... [(6, 4), (8, 4), musicmaker\_two],

... [(8, 4), (9, 4), silence\_maker],

... ]

...])

>>> all\_timespan\_lists = {

... 'Voice 1': voice\_1\_timespan\_list,

... }

>>> global\_timespan = abjad.Timespan(

... start\_offset=0,

... stop\_offset=max(\_.stop\_offset for \_ in all\_timespan\_lists.values())

...)

>>> for voice\_name, timespan\_list in all\_timespan\_lists.items():

... silences = abjad.TimespanList([global\_timespan])

... silences.extend(timespan\_list)

... silences.sort()

... silences.compute\_logical\_xor()

... for silence\_timespan in silences:

... timespan\_list.append(

... abjad.AnnotatedTimespan(

... start\_offset=silence\_timespan.start\_offset,

... stop\_offset=silence\_timespan.stop\_offset,

... annotation=MusicSpecifier(

... music\_maker=None,

... voice\_name=voice\_name,

... ),

... )

... )

... timespan\_list.sort()

>>> for voice\_name, timespan\_list in all\_timespan\_lists.items():

... shards = timespan\_list.split\_at\_offsets(bounds)

... split\_timespan\_list = abjad.TimespanList()

... for shard in shards:

... split\_timespan\_list.extend(shard)

... split\_timespan\_list.sort()

... all\_timespan\_lists[voice\_name] = timespan\_list

>>> score = abjad.Score([

... abjad.Staff(lilypond\_type='TimeSignatureContext', name='Global Context'),

... abjad.Staff([abjad.Voice(name='Voice 1')],name='Staff 1', lilypond\_type='Staff',),

... ],

...)

>>> for time\_signature in time\_signatures:

... skip = abjad.Skip(1, multiplier=(time\_signature))

... abjad.attach(time\_signature, skip)

... score['Global Context'].append(skip)

>>> def make\_container(music\_maker, durations):

... selections = music\_maker(durations)

... container = abjad.Container([])

... container.extend(selections)

... return container

>>> def key\_function(timespan):

... return timespan.annotation.music\_maker or silence\_maker

>>> for voice\_name, timespan\_list in all\_timespan\_lists.items():

... for music\_maker, grouper in itertools.groupby(

... timespan\_list,

... key=key\_function,

... ):

... durations = [timespan.duration for timespan in grouper]

... container = make\_container(music\_maker, durations)

... voice = score[voice\_name]

... voice.append(container)

>>> for voice in abjad.iterate(score['Voice 1']).components(abjad.Voice):

... for i , shard in enumerate(abjad.mutate(voice[:]).split(time\_signatures)):

... time\_signature = time\_signatures[i]

... abjad.mutate(shard).rewrite\_meter(time\_signature)

>>> for voice in abjad.select(score).components(abjad.Voice):

... for run in abjad.select(voice).runs():

... if 1 < len(run):

... specifier = abjadext.rmakers.BeamSpecifier(

... beam\_each\_division=True,

... )

... specifier(abjad.select(run))

... abjad.attach(abjad.StartBeam(), run[0])

... abjad.attach(abjad.StopBeam(), run[-1])

>>> for staff in abjad.iterate(score['Voice 1']).components(abjad.Staff):

... for rest in abjad.iterate(staff).components(abjad.Rest):

... previous\_leaf = abjad.inspect(rest).leaf(-1)

... if isinstance(previous\_leaf, abjad.Note):

... abjad.attach(abjad.StopHairpin(), rest)

... elif isinstance(previous\_leaf, abjad.Chord):

... abjad.attach(abjad.StopHairpin(), rest)

... elif isinstance(previous\_leaf, abjad.Rest):

... pass

>>> abjad.show(score)

Which, with the assistance of a stylesheet for formatting, results in the following Lilypond code and image:

\score { %! LilyPondFile

\new Score

<<

\context TimeSignatureContext = "Global Context"

{

\time 4/4

s1 \* 1

\time 5/4

s1 \* 5/4

}

\context Staff = "Staff 1"

{

\context Voice = "Voice 1"

{

{

c'16

[

cs'16

~

cs'16

d'16

~

d'8

ef'8

~

\times 4/5 {

ef'8

e'16

c'8

}

cs'8.

d'16

]

}

{

r2

}

{

e'8

[

ef'8

d'8

cs'8

]

}

{

r4

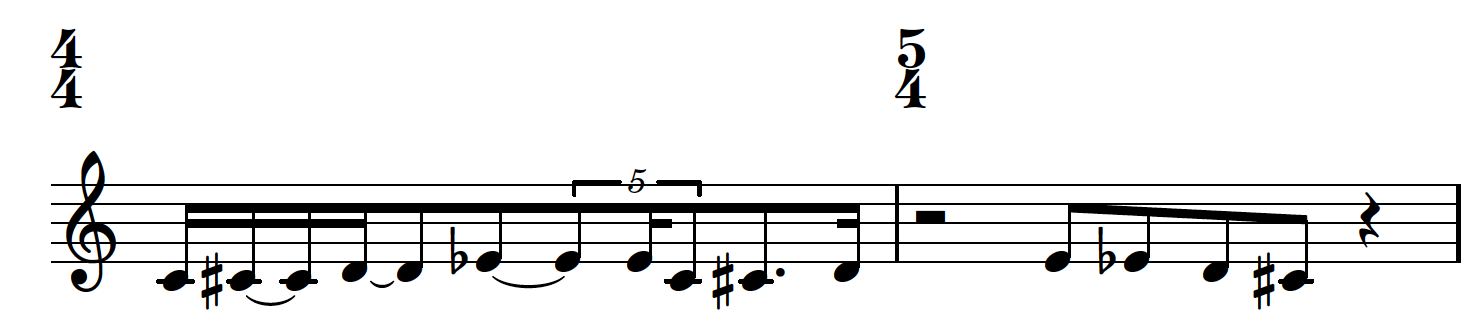
}

}

}

>>

} %! LilyPondFile



4. AttachmentHandler

AttachmentHandler is intended to work with MusicMaker, but both are able to be used independently. In essence, AttachmentHandler’s job is to attach text spanners, dynamics, hairpins, and articulations to runs of music throughout the score. When used in conjunction with MusicMaker, the composer is able to create gestures not only of distinct harmonic and rhythmic content, but they are able to sculpt a distinct dynamic and articulation profile as well. The following is the current source code for AttachmentHandler. As with MusicMaker it is still being revised, and it is being revised more extensively than MusicMaker.

import abjad

class AttachmentHandler:

def \_\_init\_\_(

self,

starting\_dynamic=None,

ending\_dynamic=None,

hairpin=None,

articulation\_list=None,

text\_list=None,

line\_style=None,

):

def cyc(lst):

count = 0

while True:

yield lst[count%len(lst)]

count += 1

self.starting\_dynamic = starting\_dynamic

self.ending\_dynamic = ending\_dynamic

self.hairpin = hairpin

self.articulation\_list = articulation\_list

self.text\_list = text\_list

self.line\_style = line\_style

self.\_cyc\_articulations = cyc(articulation\_list)

self.\_cyc\_dynamics = cyc([starting\_dynamic, ending\_dynamic])

self.\_cyc\_text = cyc(text\_list)

def \_\_call\_\_(self, selections):

return self.add\_attachments(selections)

def \_apply\_text\_and\_span\_lr(self, selections):

text = self.\_cyc\_text

for run in abjad.select(selections).runs():

leaves = abjad.select(run).leaves()

span = abjad.StartTextSpan(

command=r'\startTextSpanOne',

left\_text=abjad.Markup(next(text)).upright(),

right\_text=abjad.Markup(next(text)).upright(),

style=self.line\_style,

)

abjad.attach(span, leaves[0])

abjad.attach(abjad.StopTextSpan(command=r'\stopTextSpanOne',), leaves[-1])

def \_apply\_text\_and\_span\_l\_long(self, selections):

text = self.\_cyc\_text

for run in abjad.select(selections).runs():

leaves = abjad.select(run).leaves()

span = abjad.StartTextSpan(

command=r'\startTextSpanOne',

right\_padding=2.5,

left\_text=abjad.Markup(next(text)).upright(),

style='solid-line-with-hook',

)

last\_leaf = leaves[-1]

next\_leaf = abjad.inspect(last\_leaf).leaf(1)

abjad.attach(span, leaves[0])

abjad.attach(abjad.StopTextSpan(command=r'\stopTextSpanOne',), leaves[-1])

def \_apply\_text\_and\_span\_l\_short(self, selections):

text = self.\_cyc\_text

for run in abjad.select(selections).runs():

leaves = abjad.select(run).leaves()

span = abjad.StartTextSpan(

command=r'\startTextSpanOne',

right\_padding=2.5,

left\_text=abjad.Markup(next(text)).upright(),

style='solid-line-with-hook',

)

last\_leaf = leaves[-1]

next\_leaf = abjad.inspect(last\_leaf).leaf(1)

abjad.attach(span, leaves[0])

def add\_attachments(self, selections):

runs = abjad.select(selections).runs()

ties = abjad.select(selections).logical\_ties(pitched=True)

for run in runs:

if len(run) > 1:

leaves = abjad.select(run).leaves()

if self.starting\_dynamic != None:

abjad.attach(abjad.Dynamic(self.starting\_dynamic), leaves[0])

if self.hairpin != None:

abjad.attach(abjad.StartHairpin(self.hairpin), leaves[0])

if self.ending\_dynamic != None:

abjad.attach(abjad.Dynamic(self.ending\_dynamic), leaves[-1])

abjad.attach(abjad.StartHairpin('--'), leaves[-1])

if self.text\_list != None:

if len(self.text\_list) > 1:

self.\_apply\_text\_and\_span\_lr(run)

else:

self.\_apply\_text\_and\_span\_l\_long(run)

else:

leaves = abjad.select(run).leaves()

dynamic = next(self.\_cyc\_dynamics)

if self.starting\_dynamic != None:

if self.ending\_dynamic != None:

abjad.attach(abjad.Dynamic(dynamic), leaves[0])

else:

abjad.attach(abjad.Dynamic(self.starting\_dynamic), leaves[0])

if self.starting\_dynamic == None:

if self.ending\_dynamic != None:

abjad.attach(abjad.Dynamic(self.ending\_dynamic), leaves[0])

abjad.attach(abjad.StartHairpin('--'), leaves[0])

if self.text\_list != None:

self.\_apply\_text\_and\_span\_l\_short(run)

for tie in ties:

if len(tie) == 1:

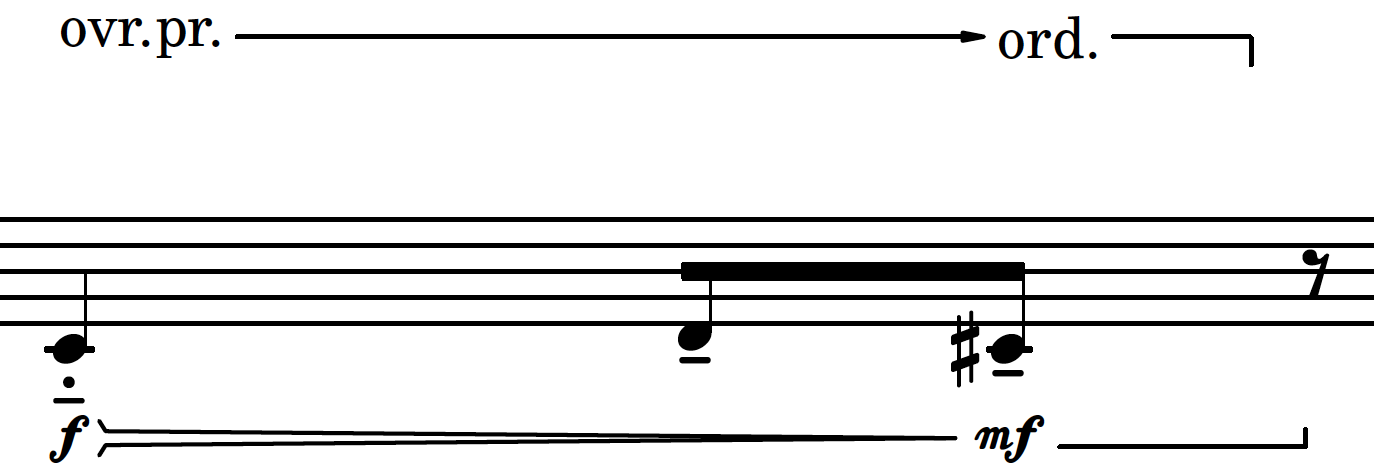
if self.articulation\_list != None:

articulation = self.\_cyc\_articulations

abjad.attach(abjad.Articulation(next(articulation)), tie[0])

return selections

It is able to produce output like the following image and is featured in all of my recent scores:



C. Back to the Source

I have just described the need for composers to write their own tools for composition with Abjad, but I have also occasionally found it necessary to edit Abjad’s source code in order to include features that I desire. This should be a rare occurrence, but is entirely possible since Abjad is open source.

1. Clef.py

Recently I edited the *Clef.py* file in the Abjad source. I did this in order to include Abjad representations of clefs that were present in the most recent update of Lilypond. The clefs in question were *varC* and *tenorvarC*, both of which are alternative c clefs. I wanted to add these clef for a logical reason as well as a very vain reason. The first reason is that, as much as possible, Abjad should have a representation of all of Lilypond’s features. If a composer knows that Lilypond is capable of producing a certain graphic object, it can be very frustrating to find no way to use it in Abjad. The second reason is that these clefs more closely represent my own handwriting of c clefs than the traditional c clef, so it feels very comfortable to me and I want to use it in my scores!

2. Articulation.py

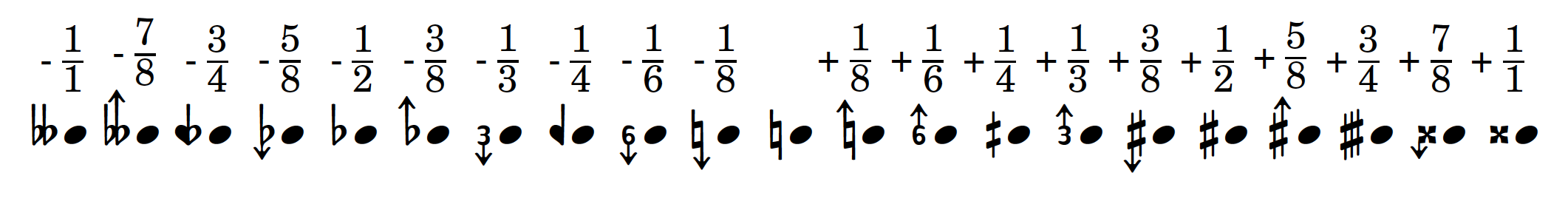
Just as with *Clef.py* I also recently edited the *Articulation.py* file. I did this at the same time and for the same reason as editing the clef file. I added Abjad representations of *halfopen* a circle with a diagonal slash and *snappizzicato* which is the common notation for a snap or “Bartok” pizzicato.

3. Microtonal Expansion in Abjad 2.21

In the summer of 2018, I undertook, with help from Ivan Alexander Moscotta, a much larger revision of Abjad’s source code. These edits were specifically centered around Abjad’s representation of pitch. At the time, the most recent version of Abjad was Abjad 2.21. Also during this summer, I attended the CCRMA Abjad workshop and I was able to discuss some of these changes with the primary maintainers of the system: Trevor Bača and Josiah Wolf Oberholtzer. We came to the conclusion that much of Abjad’s representation of microtones should be reassessed and should be open enough for composers to be able to define their own accidentals and scales. Because of this decision, the changes that I made to Abjad 2.21’s code is not available for users in Abjad 3.1, but will hopefully be given new birth in a future release.

a. Why?

I decided to undertake this major revision because I wanted to compose with microtones smaller than Lilypond’s and Abjad’s smallest interval: the quarter tone. In Lilypond’s font, *Emmantaler*, are two different kinds of quarter tones. Quarter tones written in Stein-Ellis notation and quarter tones written as traditional accidentals with an attached arrow either up or down to represent the microtonal alteration. I decided to use these arrow-based quarter tones to represent eighth tones. In fact, there is a file buried deep within Lilypond called Microtonal.ily that does just this! The file must be included at the header of the Lilypond file in order to make use of the user-defined microtones. I tried to find a way to do this and had a little success, although with great difficulty. I began to wonder if it was possible to extend this to further divisions of the octave. I began to edit the default font in Lilypond to be able to represent different kinds of accidentals as well as making some slight changes to the default accidentals for my own graphic preference, generally keeping to the Stein-Zimmer notation for quarter tones and the Ferneyhough notation for all other microtonal alterations. This became cumbersome and inconsistent and I looked for an alternative. Fortuitously, I found the Ekmelily system. This extension of Lilypond, written by Thomas Richter, does something similar to Microtonal.ily, but it also comes with an extensive font extension to allow for many kinds of microtonal representations and the ability to create user-defined scales with accidentals chosen by the user. This was my solution for graphically representing my new microtones. The following image is a representation of my own user-defined scale:

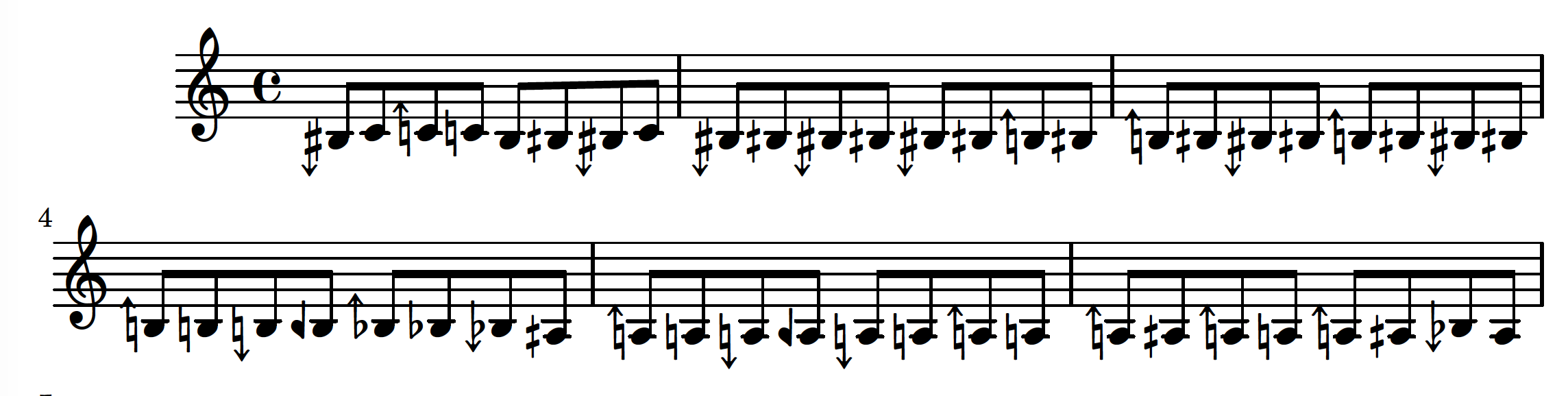


b. file systems and alterations

There were a few files in Abjad that needed to be changed in order to interface with Ekmelily via Abjad. These files were *Accidental.py, NumberedPitchClass.py,* *PitchClass.py,* and *language\_pitch\_names.py*. I also edited the *language\_pitch\_names.ly* file in Lilypond. In these files I defined the name, division size, and abbreviation of each new accidental and linked these abbreviations to my user-defined scale in Ekmelily. Making sure to always include Ekmelily and my own scale at the beginning of each Lilypond file, I was able to compose music in abjad with eighth tones, third tones, and sixth tones. All of the code alterations for this functionality is available at [*https://github.com/GregoryREvans/Abjad-Microtones*](https://github.com/GregoryREvans/Abjad-Microtones)*.*

c. abjad 3.0 system change and removal of float paradigm

During the summer of 2018, Abjad 3.0 was released. In this version of Abjad, many things were changed in the underlying system that made the porting of my microtonal edits to Abjad 3.0 difficult. During this time, I discussed the possibility of added my functionality to Abjad and Trevor Bača, Josiah Wolf Oberholtzer, and I decided that it might be best to represent microtones as something other than floating point decimals. Work was begun on a new pitch system for Abjad, but this addition to Abjad has yet to be completed. When my microtonal edits to the Abjad source code were up to date with the most recent Abjad release, I was able to produce material like the following eighth tone random walk:



D. Conclusion

In this chapter, we have seen my methodology for composing in Abjad and the tools that I have written to assist in my compositional process. In the next chapter I will present the source code and scores of recent music that I have written with Abjad, all of which is available at [*https://github.com/GregoryREvans*](https://github.com/GregoryREvans).