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, on the other hand, don't make mistakes unless they are taught a false procedure. Computers are machines and don’t even 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 by 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, 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.

A great advantage to the Abjad composition paradigm is its ability to manage polyphony. Other programming paradigms like OpenMusic or PWGL are a little more restricted in this regard. Often, in OM and PWGL, continuing a procedure from one instrument to the next is more akin 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 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 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 OM and PWGL 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 important features of Abjad that convinced me to use it is that Abjad 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 abstractions. 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. Both OM and PWGL are based on the LISP programming language, but I believe that the object-oriented nature of Python makes it a much better candidate for modelling music notation. 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

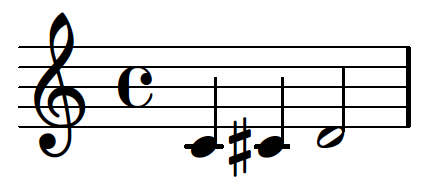
cs'4

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

{

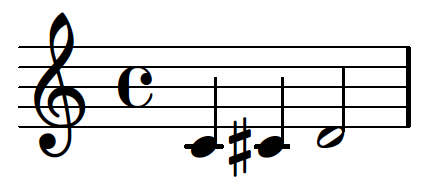
c'4

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! Next, I will address howwe might write a procedure to handle the *abjad.BowContactPoint()* object, which produces a more complex Lilypond result and graphic.

b: 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.

c: dynamics, articulations, and hairpins

Just like the creation of note objects and attaching bow contact indicators, 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

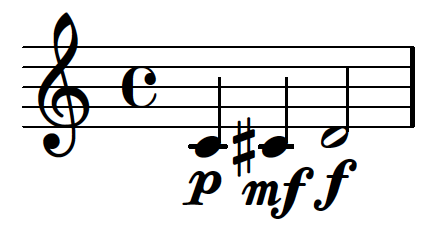
\mf

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

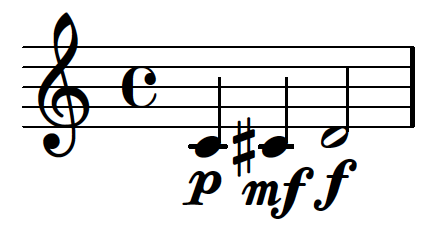
\mf

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

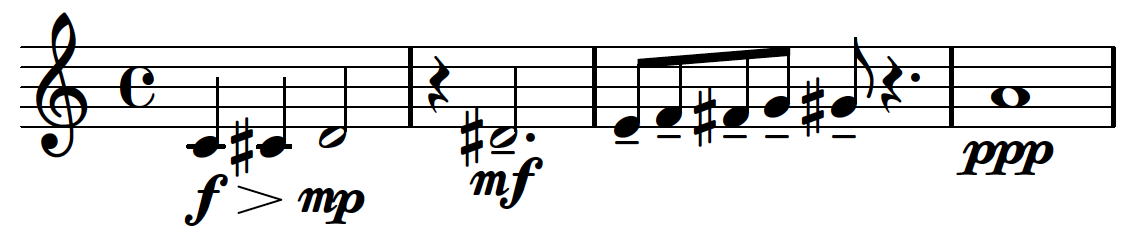
r4

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.

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 = list()

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 moving in step of plus or minus 0.5 that are turned into note objects that are placed in a staff. When the staff is shown, the code 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 that end before the slowest are repeated until the bottom voice is completed. 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 i build my own tools?

2: abjad-ext

a: why does abjad-ext exist?

b: rmakers

c: other packages

3: 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[:]]

4: AttachmentHandler

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

C: Back to the Source

1: Clef

2: Articulation

3: micotonal expansion in abjad 2.21

a: file systems and alterations

b: microtonal.ily

c: editing the font

d: ekmel.ily

e: abjad 3.0 system change

f: removal of float paradigm

g: in progress