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 beautiful 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 instrumentation, 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 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 potentials. 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. The objects of notational elements are manipulable and 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 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 be used just as easily 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. As with any attempt to summarize behavioral activity with computational processes, these solutions are a work in progress and may not necessarily have universal functionality, but I have found a great amount of use out of the programs that I have written. 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 do I 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 few 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

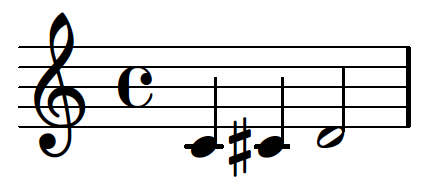
cs'4

d'2

}

} %! LilyPondFile

and image:



As you might begin to suspect, this process of note creation can get quite tedious. Why don’t we look at a way to write code in abjad that gives us the exact same output in our Lilypond and image files? Here is a possible alternative:

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

{

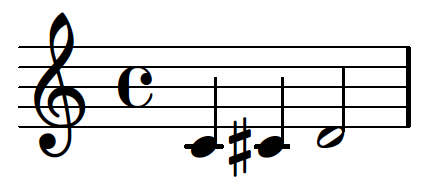
c'4

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, let’s look at how we might write a procedure to handle the *abjad.BowContactPoint* object.

b: Using BowContactPoint

The *abjad.BowContactPoint* object and an accompanying factory class 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 this tool:

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 *\tweak* commands in the Lilypond code. Imagine composing a score 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. Let’s look at a few alternative methods that achieve the same results in a similar manner of reduction as in the note creation example:

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 here 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 results in 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 them to each leaf of the staff. Notice that each version of our code results in the same output, but each option simplifies the process.

c: dynamics and hairpins (show abjad, show Lilypond, show image)

d: abjad.Select, abjad.Iterate

e: stylesheets

These are just a few examples of ways in which abjad allows 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 looked at the power that Python gives us, let us look at how creating these loops and functions has further ramifications in the process of composing.**

3: Composing in an Iterable Fashion

4: Composing with Algorithms and Models

5: A Comfortable Formalism in Score Control is Not Necessarily Algorithmic Utopia

This process should not be misperceived as a purely algorithmic system for music composition. Certainly, formalizing elements in a score

6: The Challenges of Idiomatic Writing

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: 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

4: MusicMaker

5: AttachmentHandler

C: Back to the Source

Clef

Articulation