Chapter I. Some Prerequisite Knowledge of Lilypond and Python

In my experience composing scores with the help of computational systems, I have found that the greatest power and flexibility are available from the Abjad Application Programming Interface for formalized score control. Abjad is significant because of the freedom with which the composer is able to manipulate their musical material and the ability to not only control the musical elements of a score, but also other graphic features as well. When choosing to compose music with Abjad, there are some elements of the underlying software with which the musician should familiarize themselves. Every score that is created with Abjad is engraved by the Lilypond music notation engine. Because of this interdependence, the composer should be familiar with Lilypond’s model of music notation as well as elements of Lilypond syntax. Since Abjad is an API in the python programming language, it is essential that the composer be familiar with writing python code. In this chapter, the basics of Lilypond and Python will be discussed, while information directly related to the Abjad API follows in chapter two.

A: Lilypond’s model of music Notation

Any modern composer should be familiar with the plethora of options available for digital music engraving. The purpose of this paper is not to relate the history of modern engraving practices, but it is important to note that, by far, the most popular engraving software available for the consumer market today are Finale and Sibelius, with a few robust newcomers like Dorico beginning their careers. These systems, packed with feature after feature, are suitable for the majority of composers’ needs. They are able to engrave pitches and rhythms in traditional Western notation and allow for a number of formatting options that expand upon these traditions and allow the user to create professional-quality documents, but it is not insignificant to note that many composers find the musical model of these programs to be overly restrictive upon musical creativity. As an example, in most of the common engraving software, it takes 5 or more extra clicks of a mouse through a few menus to engrave a tuplet other than a triplet. Programs like Dorico have supplemented some of these issues through keyboard shortcuts and opened a clearer accessibility to the engraving of insected tuplets, but otherwise it is clear: these programs are tailored to a specific set of engraving requirements. This software is made for people engraving fairly traditional music like transcriptions, orchestrations, film scores, and so on.

While the programs are flexible and can be used for other means, many composers have decided that what they require from a musical score is significantly restricted by the software. It becomes tedious to write music with many tuplets or other graphical oddities. Some composers have written their own engraving engines, like NoteAbility Pro, which can handle a number of contemporary techniques with ease, but other composers have resorted to simply composing in graphic editors and drawing-oriented software, which brings the act of engraving much closer to the act of handwriting a sketch or a full composition, but even with these paradigm shifts, few notation engines show any friendliness to structural formalization. Finale and Sibelius have features that allow the user to program certain procedures, but these are limited. Programs like Patch Work (and its kin Patch Work Graphic Language) and OpenMusic were created in order to fill this gap. These programs allow the composer to manipulate data to represent musical elements which are then engraved within the program, in the case of PWGL, or exported as MusicXML, in the case of OpenMusic, to be engraved by another software, but MusicXML is not the most stable of protocols and often produces fallacious results or completely fails to convert in many instances.

The combination of Abjad and Lilypond surmounts all of these concerns. Abjad simply writes text files of Lilypond code, which removes the concern of file transfer errors, and Lilypond represents each element of a score, music, text, or graphic, in a syntax that is simple and consistent across a number of engraving complexities, allowing the composer to engrave almost anything as part of the score. Another important aspect of Lilypond is its modelling of rhythmic content. Lilypond makes a distinction, unlike other notation engines, between written and prolated durations. In programs like OpenMusic, a set of triplet eighth notes would be written as durations of 1/3, 1/3, 1/3. But in Lilypond they would be written as 1/8, 1/8, 1/8 prolated by a duration of 1/3. This means that traditional rhythmic notation like whole notes through sixty-fourth notes and beyond are written as usual, but are *prolated* by a surrounding tuplet bracket with a given duration. Abjad follows, as much as possible, the same conventions as Lilypond’s notational model.

Lilypond also has a feature referred to as “Context Concatenation.” A context in Lilypond can be thought of as a staff with various features and formats associated with it. When given a name, a context is able to be appended to another with the same name. This allows the composer to write various sections of a piece in isolation and to stitch them together into a final document as a secondary process. Users should note a similarity between Lilypond syntax and LaTeX syntax, both of which are conceptually similar to HTML code. There are a number of Lilypond tutorials to be found on the internet, so space will not be wasted here as yet another Lilypond Tutorial.

B: Python

In Python, there are a number of data types. Some of the prominent data types with which we will be dealing when using Abjad are integers, floating-point decimals, variables, strings, lists, tuples, and dictionaries. Each of these data types have specific features and behave in certain ways. Both integers and floating-point decimals, often called floats, are numbers. Readers familiar with arithmetic will recognize these distinctions. Integers can be used to signify numeric value in whole numbers while floats offer a more refined value. Variables, much like in algebra, are names that are assigned to other values or processes. With variables, we are able to refer to data sets or processes throughout a file without rewriting the information many times by hand. The process of programming can be crudely summed up as the manipulation of data via computational means.

An important process to comprehend when composing with abjad is that of list manipulation. There are many processes that can be performed on and with lists, but first we should discuss the concept of slicing. Readers vaguely familiar with Python will recognize the format [x:y] when referring to slicing a list. In Python, we can refer to items within a list via their indices. The index is the location within a list where an item exists. These indices begin at zero. An example set of indices is [0, 1, 2, 3, 4, 5 …], but the Python slices [x:y] do not refer to items, even though indices do refer to items. The indicators within a slice actually refer to the spaces between items. We can test this principle as follows:

>>> list = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]

>>> print(list[2])

Which results in

‘c’

but:

>>> list = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]

>>> print(list[0:2])

results in:

‘a’, ‘b’

Now let’s try the following:

>>> list = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]

>>> list[-1:1] = ‘xyz’

>>> print(list)

Which will result in:

[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘x’, ‘y’, ‘z’, ‘f’]

We see, in fact, that this slicing refers to the continuous space between -1 and -1.

Another of the many actions upon lists that are able to be performed on lists is that of list comprehension. List comprehensions allow the programmer to quickly create lists whose contents follow simple parameters. We have the built-in function range(). range() allows the user to increment integers up until the user-input point. If Python were asked to print each item within range(5), then 0, 1, 2, 3, and 4 would be written to the terminal. We could write a list comprehension such as the following:

>>> list = [x for x in range(5)]

>>> print(list)

Which will result in:

[0, 1, 2, 3, 4]

We can also act upon the elements within this list:

>>> list = [x\*3 for x in range(5)]

>>> print(list)

Which will result in:

[0, 3, 6, 9, 12]

We can also write this process as a “for loop”:

>>> incremets = range(5)

>>> list = []

>>> for x in increments:

... list.append(x\*3)

>>> print(list)

In this example we can see the use of append(). It is important to make a distinction between append() and extend(). This can be illustrated as follows:

>>> list\_1 = [0, 1, 2, 3]

>>> list\_2 = [4, 5, 6, 7]

>>> list\_1.append(list\_2)

>>> print(list\_1)

Which results in:

[0, 1, 2, 3, [4, 5, 6, 7]]

but:

>>> list\_1 = [0, 1, 2, 3]

>>> list\_2 = [4, 5, 6, 7]

>>> list\_1.extend(list\_2)

>>> print(list\_1)

results in:

[0, 1, 2, 3, 4, 5, 6, 7]