Dmitri Tymoczko
Princeton University
June 2013

MTO REVIEW OF MUSIC21

Music21<LINK http://web.mit.edu/music21/> is Michael Cuthbert's library of programs ("modules") allowing the Python computer language to read, write, and manipulate musical scores. Having used it for about a year, I am convinced that it will play an important role in music theory's future. I strongly recommend it to anyone interested in algorithmic composition, computer-assisted analysis (of both single scores and larger corpora), or what could be called "bog-standard music-theoretical computation" (e.g. "find those equal-tempered scales, with less than 45 notes, containing all-interval sets, and list those sets"). Indeed, the first piece of advice I give to beginning theory graduate students is "learn music21." We may be waiting for our Sigfried, but we have already found his Nothung.

Click here for SIDEBAR 1, a very brief introduction to Python.

It is, to be sure, also a source of frustration: for the foreseeable future music21 will remain a work in progress, somewhat in the way that medieval cathedrals were built over centuries. But there is a growing and supportive community of users, and Cuthbert, who leads a team of paid, student, and volunteer software developers, is incredibly responsive to that community's needs, producing software updates almost monthly. (The documentation has also been extensively revised since I learned the program.) In a real sense, to use music21 is to collaborate in its development. For those who are patient enough to deal with the occasional bug or unexpected behavior, it provides unparalleled tools for analyzing and creating music.

What can you do with music21? Let me answer by describing what I have done in the last year. (1) I built a Roman numeral analyzer that can analyze simple tonal (or quasi-tonal) scores such as Bach chorales and Morley madrigals, using it to analyze hundreds of scores freely available on the internet. I then wrote a different music21 program to compare the first program's output to my own handmade analyses, determining that my analyzer is about 82% accurate. (Which is to say that it gets the key right about 90% of the time, and, given the correct key, gets the chord right 90% of the time.) A third program proofreads my analyses, identifying those places where the pitches indicated by the Roman numeral are not found in the score. (2) I wrote a program that, given a score and a Roman numeral analysis (including one produced by my program), can label the nonharmonic tones—or produce a new score in which these tones have been replaced by the appropriate harmonic tones. (3) I wrote a series of programs that combine information from scores and harmonic analyses, so that I can ask (a) which notes are doubled in particular chords ("does Bach typically double the bass in a first-inversion triad?"), (b) what voice leadings are used with particular progressions (e.g. "how does scale degree 4 typically move in vii$^{\circ 6}$–I?"), (c) what sorts of parallels appear on various reductional levels (including the musical surface and the "reduced" scores

produced by removing nonharmonic tones), and various other related questions.[1] (4) I've used it in various compositional contexts, constructing scores that exemplify interesting musical processes (e.g. fractal music based on the logistic equation, or infinite canons based on Cliff Callender's brilliant musical interpretation of the logarithmic spiral[2]). If these are projects that appeal to you, then music21 is the sort of thing you will like.

I think of music21 as being composed of two parts. The first is infrastructure, routines for reading, writing, and manipulating musical scores, while the second consists of a higher-level analytical toolkit—generating a Roman numeral from a chord and key, putting chords into normal form, checking for parallel fifths, identifying scales containing a given pitch or chord, and so on. Of the two, the first is in my opinion the most valuable. In large part, this is due to the inherent strength of Python itself—given the basic infrastructure in music21 it is not terribly hard to write your own routines to put chords into normal form or identify them by Roman numeral. In fact, you should expect to do some of this, since (for instance) your desired normal-form algorithm may diverge from the program's "out-of-the-box" behavior. For instance: should the algorithm handle floating-point numbers or just integers? How should mutisets be treated? Which precise variant of normal-forming should be used? No toolkit can satisfy every task, and to the extent that it tries it will become bloated. More on that in a moment.

The great thing about music21, from the standpoint of infrastructure, is its ability to understand all the common formats for representing musical data. First and foremost, it can read and write musicxml, which serves as a kind of "lingua franca" spoken by all the major notation programs. This means that music21 can return its results *as music,* constructing new scores that (for instance) contain all examples of a specific harmonic or melodic progression in a particular corpus; it can also store its "data" (which is to say, pieces of music) as musicxml files, which can then be printed and edited just like any other score. (Music21 can also use Python's "pickling" capability to store its own internal data structures.) Beyond this, music21 can read MIDI files, "kern files" (David Huron's older text-based format, which has been used to encode a large corpus of scores—mostly available online); "ABC notation" (another notation format, popular for folk music) my own "romantext" format for encoding Roman numeral harmonic analyses; David Temperley's variant format used in his popular-music corpus; and a good many other formats besides. Since music21 users can access the immense amount of symbolic music data on the web, large-scale corpus analysis is suddenly much, much easier.[3] (Not too long ago, corpus studies required an enormous amount of data entry,

---

[1] The answers to these questions are: (a) Bach usually doubles the bass note, even in first inversion triads (except V^6, of course); (b) in vii°^{6}–I, scale degree 4 typically moves upward to scale degree 5, rather than "resolving" to the third; and (c) for the most part, parallels appear in the background only when they are masked by suspensions at the foreground level.

[2] Paper presented to the national meeting of the Society for Music Theory, New Orleans, 2012.

[3] In fact, the program can even load scores by URL, so you don't have to keep the data on your hard drive.

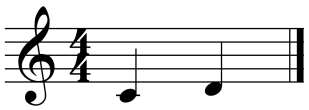
and often utilized project-specific formats—limiting your ability to share this data with others.) This flexibility is a tremendous triumph for Cuthbert and collaborators, exemplifying the open, idealistic pluralism of the internet's higher self.

The infrastructure also provides basic tools for creating and manipulating musical information: music21 has data structures representing pitches, notes (pitches combined with durations), chords, scales, keys, text expressions, dynamic expressions, clefs, figured-bass parts, twelve-tone rows, measures, musical parts, and many other concepts as well. Python (and hence music21) is an "object oriented" language, so these data structures are bundled together with operations that can be applied to them. Thus, for example, various objects—from pitches to key signatures—can be transposed by calling their ".transpose" method. These methods are generally quite comprehensive, and I have always found myself able to achieve my goals, though not always as quickly as I might like.

The fundamental object of music21 is the "stream," an extremely abstract data structure that allows (basically) any piece of information to be stored at any particular "offset," or point in time. (In general, offsets are measured in quarter-note units, though this interpretation is not enforced by the program.) Thus for example, I can make a simple score as follows:

```
>>> from music21 import *          # import all the music21 routines
>>> s = stream.Stream()            # create a new stream and call it "s"
>>> s.insert(0, note.Note('C4'))   # insert Note C4 at offset 0 in s
>>> s.insert(1, note.Note('D4'))   # insert Note D4 at offset 1 in s
>>> s.show()                       # show the result
```

(Click here for SIDEBAR 2, on Python conventions.) This opens my notation program (Finale) and displays the following score.



(Note that music21 has added some defaults: a time signature of 4/4, and quarter-length durations for each note. In real applications, I would typically provide this information myself.) Alternatively, I could examine the underlying data by showing the music21 stream in text format:

```
>>> s.show('text')    # call the "show" method with parameter 'text'
```

which prints out:

```
{0.0} <music21.note.Note C>
{1.0} <music21.note.Note D>
```

Indicating that the stream has the note C at offset 0 and the note D at offset 1.

Learning music21 in large part involves coming to terms with the idiosyncracies of the "Stream" object. In my experience, there are four. First, a single musical object (note, measure, etc.) can belong to multiple streams. While this flexibility is in principle admirable, it can lead to unexpected behavior:

```
>>> n1 = note.Note('C4')     # create a Note "C4"
>>> s1 = stream.Stream()     # create an empty Stream
>>> s1.insert(0, n1)         # insert the note at offset 0
>>> print n1.offset          # get its offset
0.0
>>> s2 = stream.Stream()     # create a second empty stream
>>> s2.insert(5, n1)         # insert the same note at offset 5
>>> print n1.offset          # check its offset
5.0
```

The issue here is that there is a fundamental difference between "absolute" attributes of the object (such as the pitch of some note, or the length of a duration) and "context-dependent" attributes such as its offset. (This difference was not clearly explained in the documentation when I learned the program.) Technically, music21 objects have an "activeSite" attribute that points to the Stream in which they are currently considered to reside; inserting a note into a new Stream changes its context or activeSite. Learning to manipulate the activeSite is fundamental, but rather obscurely documented:

```
>>> n1.activeSite = s1       # set the activeSite to stream s1
>>> print n1.offset          # check note n1's offset
0.0                          # aha! Back to 0!
>>> n1.activeSite == s2      # is n1's activeSite stream s2?
False
>>> n1.activeSite == s1      # is it s1?
True
>>> n1.activeSite = s2       # change the activeSite to s2
>>> print n1.offset          # check the offset now
5.0
```

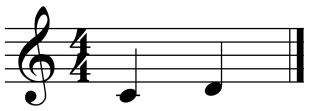
In the bad old days, built-in music21 routines would change a note's activeSite without warning, creating extremely hard-to-fix bugs. That problem now seems to have been solved.

Second, streams are typically massively hierarchical, composed of many "substreams" nested inside one another. Thus a "Score" (a kind of stream), will contain "Parts" (another kind of stream), which will contain "Measures" (yet another kind of stream), which may have "Voices" (a kind of stream for dealing with multiple lines of music on a single staff, analogous to Finale's "layers" or "voices") which will themselves

have notes, rhythms, clefs, time signatures, and all the other basic data you are interested in. To find a particular musical object, you need to access the right level of hierarchy, and this can take some work. (Click SIDEBAR 3 for an example.) Music21 has a method for "flattening" streams, but this creates its own problems, since it returns a *new* stream rather than giving you access to the original, hierarchical version—thereby inviting all the complications of the previous paragraph. As of this writing, music21 is transitioning toward a new method of dealing with hierarchy (involving the Stream.recurse method) but this is still in progress. In any case, this massively imbricated hierarchy can confuse even experienced users.

A third issue is that that music21 stores data in its own abstract format, but typically communicates with the outside world by way of musicxml. The divergence between internal data and musicxml can lead to confusion, as in the following code.

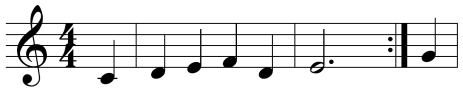
```
>>> from music21 import *          # import all the music21 routines
>>> s = stream.Stream()            # create a new stream and call it "s"
>>> s.insert(0, note.Note('C4'))   # insert Note C4 at offset 0 in s
>>> s.insert(2, note.Note('D4'))   # insert Note D4 at offset 2 in s
>>> s.show('text')                 # show the result as text
{0.0} <music21.note.Note C>
{2.0} <music21.note.Note D>
>>> s.show()                       # show the result in notation
```



The problem is that our score has a "gap," an empty space between the end of the first note and the beginning of the second. (Since we didn't give our notes any duration, they have been assigned the default length of one quarter note, which means that the stream contains nothing in the beat between the end of the first note and the start of the second.) This "gap" is perfectly acceptable to music21, but not to notation programs such as Finale. Accordingly, somewhere in the translation process from music21's internal representation to musicxml (as interpreted by Finale), the note gets moved from the third beat to the second. (Perhaps a better default behavior would be for music21 to insert the appropriate rest, or extend the duration of the first note.) One can spend many frustrating hours wondering why a note that is at offset 2, in music21, shows up in Finale at offset 1. The answer is that music21 translates its own internal representation into musicxml before sending it out to the notation program.

The final issue is in many ways the opposite of this one. While music21 in principle has its own internal data format, in which almost any data can be placed at any offset, in practice it tends to hew pretty closely to the musicxml specification. Consequently, music is not necessarily stored in the way a theorist would expect. Take

for instance, the vexing matter of mid-measure repeats. To the musician, the following passage has a repeat bar occurring three quarters of the way through measure 2:



But in music21 (as in all the major notation programs) this passage involves four separate measures. Measure 0 is a 1/4 "pickup" measure not included in the numbering scheme; measure 1 is a standard 4/4 measure. Measure 2 is a 3/4 measure with a repeat bar at the end, in which the time signature is suppressed. It is then followed by an unnumbered 1/4 measure with suppressed time signature. All of this is reasonable enough from the standpoint of a notation program's private data format, not meant to be directly accessible to users, but it is a true headache for the analyst. Essentially it means that one cannot simply "retrieve measure 2" (or figure out what measure of a score offset 8 is found in) without a fair amount of computation. I have worked with Cuthbert to try to build some useful routines here, but there is still more to do. (Cuthbert himself wants to support mid-measure repeats eventually, bringing music21 into line with the conceptual structure of the music.) Ironically, then, this is a case there music21 uses an internal representation that, for a theorist, is *too close* to musicxml. It would be more convenient to have the internal data represent the logical structure of the music, rather than the pragmatic needs of a notation program.

This brings me to what I consider to be the three major issues with music21. First, the documentation can be somewhat spotty, and particularly challenging for those who are new to Python or object-oriented programming languages more generally. (I would have benefitted greatly, when I was starting, from a brief review such as this.) Writing code is more fun than documenting it, and as a result crucial bits of the program remain undescribed. (The documentation is full of endearing notes like "TO DO: EXPLAIN THIS" or, more ominously, "TO DO: BUG.") Since Python is an interpreted language, one can always read the source code, but that will be hardest for those beginners who need the most help. For newcomers, the music21 email list is the place to go.

My second complaint is that music21 can be very slow. Certain operations, such as reading a score or processing Roman numerals, will take much longer than you expect. Consider, for example, the simple task of identifying a scale containing a given set of pitches. Music21 has a built-in method to accomplish this task (the deriveAll method of the ConcreteScale class), but I found it too slow for my purposes; as a result, I ended up writing a streamlined routine that works much faster. (Granted, this was not too hard—my code, shown in SIDEBAR 4, involves all of 25 lines, and could be compressed even further.) The deeper issue here is that there is an inherent trade-off between flexibility and speed, and Cuthbert, in designing music21, has almost always erred on the side of flexibility. As a result, one often needs to leave the computer running overnight when processing large amounts of data.[4]

---

[4] To be sure, there are various other ways of speeding up music21, including caching data (facilitated by Python "pickling"), installing faster versions of Python such as PyPy, or

My final, and partially aesthetic, complaint is that the program has a slightly cluttered feel. As I said, for me the chief value of music21 involves its core functions of reading, writing, and performing basic manipulations on musical scores. But music21 also contains countless somewhat cutesy music-theoretical tools. (Many of these, one senses, arose as projects manageable enough to be given to undergraduate assistants.) In my experience, a number of the bells and whistles do not ring or blow as advertised—for example, an early version of the software promised to translate between various numbering systems for the Bach chorales, but used a very partial list of chorales; similarly, the early version of the "romanNumeralFromChord" routine would often produce incomprehensible gibberish. (As of this writing, the list of chorales remains somewhat incomplete while the program continues to have trouble parsing Roman numerals for augmented sixths.) From the user's point of view, incomplete or semi-functional routines are often worse than nothing at all: I have wasted hours writing code, on the assumption that a certain music21 routine would work as advertised, only to discover that I was relying on semifunctional software. This leads to tedious detours where one rewrites some part of music21 so that it works right.[5]

In general, then, I have often found myself wishing that music21 sacrificed some of its breadth, and that more attention had been spent making the core routines faster, more reliable, and completely documented. Related to this is the worry that some of music21's flexibility is more notional than real: while the program has been designed for the widest possible range of users, including twelve-tone and minimalist composers, I suspect that its current body of real-life users are more circumscribed in their interests, with corpus analysis the most common activity. It might have been nice to optimize the core routines first, building out the more specialized routines as the need arose.

Having said that, I feel obligated to recognize Cuthbert's enormous and selfless labors. A project like music21 fits uncomfortably into our standard scholarly categories: inherently not subject to peer review, it is the product of the same effort that would be required to produce a gigantic scholarly monograph (or perhaps several of the same). Yet tools like music21 are unquestionably central to the future of music theory, enormously expanding the scope of music-theoretical possibility. It is, consequently, critical that our field find ways of supporting these efforts. MIT, where Cuthbert teaches, is to be commended for sheltering this vital, twenty-first-century work.[6]

---

utilizing cluster-computing. Still, I worry that some core routines take longer than they should.

[5] Ironically, the original version of the code in SIDEBAR 4 provides an example: because of a known bug with the interval.subtract method, the routine would not work if an input scale contained an augmented unison as the second note. However, the closely related "interval.complement" turned out to be bug-free.

[6] Worse yet, free software development is inherently a slightly masochistic endeavor. Software is something that needs to *work*, and users necessarily depend on it in a way they don't depend on "mere" scholarly writing; as a result, they often respond to bugs with grumpiness and frustration, despite the fact that they haven't paid for the program.

Twenty years ago, David Huron released his "humdrum toolkit," an early set of tools and data structures for analyzing music. This work, while wonderful in its way, now feel somewhat dated—particular in comparison to modern successors like music21. The switch from Unix scripts to Python is in itself a substantial improvement, providing a powerful and readable cross-platform programming environment. Equally significant is the advent of musicxml, which allows music21 to function as part of an integrated ecosystem that includes notation programs, internet databases, software samplers, and virtually all the other tools of contemporary computer music. These tools have effectively lowered the barriers to a whole range of analytical and compositional activities, allowing users to access the enormous amount of music on the web. One hopes that this will in turn open the doors for a new generation of thinkers who are equally sophisticated musically, theoretically, and computationally. Indeed, an unintended consequence of Cuthbert's work is to forge a new road into music theory, one that begins in computer science rather than composition or performance.

[SIDEBAR 1].

Python is a very popular and relatively recent programming language, available for free for virtually any operating system, and—conveniently enough—preinstalled on Macintosh computers. It is a high-level interpreted language, which means it is convenient but slow. Python contains powerful tools for manipulating complex data structures such as lists and dictionaries, all of which can be recursively nested, so that one can construct (for example) very large lists of dictionaries of dictionaries of lists. Best of all, Python has extensions that can do virtually anything you would ever want to do. Do you want your program to send you an email after it has analyzed all the Bach chorales? Do you need access to the operating system's clipboard? Or to calculate some statistical functions? Would you like to make a graph of some interesting musical statistics, save the graph as a PDF, and open that PDF in some other application? There are libraries to do all of these things, and virtually anything else you can imagine.

Like happy families, computer programming languages are all essentially the same; users will therefore have little difficulty learning Python if they are familiar with some language like Basic, Pascal, or C. There are however a few Pythonistic idiosyncacies. The first, and most notorious, is that in Python, *indentation is syntactical, determining the scope of definitions and loops*. As a result the following lines of code produce different results:

```
c = 0
for i in range(0, 5):
  print i
  c = c + i
print c
```

```
c = 0
for i in range(0, 5):
  print i
c = c + i
print c
```

In the code on the left, the addition is within the scope of the loop, and the program leaves the variable *c* equal to 10, or 0 + 1 + 2 + 3 + 4 (ranges in Python do not include their upper bounds). In the code on the right, the addition is outside of the loop, and the program leaves *c* with a value of 4 (the highest value of *i* attained in the loop). This takes some getting used to, since programmers are accustomed to thinking of whitespace as meaningless; indentation errors can be the source of hard-to-find bugs.

A second unusual feature of Python is that assignment statements are always treated *referentially*. That is, a statement like "a = b" does not typically *copy* the contents of *b* to the memory location *a,* but rather leaves *a* pointing to the same memory location pointed to by *b*. This can lead to some nonintuitive behavior, as in:

```
l = [1, 2, 3, 4]
m = l
l.append(5)
print m
```

```
l = [1, 2, 3, 4]
m = l
l = l + [5]
print m
```

The code on the left prints [1, 2, 3, 4, 5], while the seemingly identical code on the right prints [1, 2, 3, 4]. On the left, what happens is that the variable *m* points at the memory

location pointed to by *l*, which contains the list [1, 2, 3, 4]. That list is then extended via the statement "l.append," whose result is deposited at the same location in memory. When we examine this location with "print m" we see the longer list. By contrast, the code on the right, having pointed *m* to the memory location containing [1, 2, 3, 4] then creates a *new* list, in a new memory location, by appending the list [5] to [1, 2, 3, 4], pointing the label *l* to this new location. (The different behavior of the .append method and the "+" operation is idiomatic, something you just have to learn.) This new pointing-of-l does not affect the previous pointing-of-m. Hence, when we print *m* we are looking at the memory location containing the previous list [1, 2, 3, 4].

[SIDEBAR 2]

Python has a number of data types including integers, floating-point numbers, strings, lists and dictionaries. Lists are identified by square brackets and are referenced by order position, numbering from 0:

```
>>> myList = [1, 2, 'apple', 4]
>>> print myList[3]
4
```

Dictionaries, which are identified by curly braces, are a bit like lists, except their elements are referenced by a *key* rather than an index.

```
>>> myDict = {'firstKey': 1, 'secondKey': 2, 3: 3, 'forgottenThing': 'HELP!'}
>>> print myDict['firstKey']
1
```

Lists and dictionaries can be embedded within one another, leading to very complex data structures. For instance, in my work I record the locations of all the different voice-leadings in the Bach chorales; this involves a list containing dictionaries whose elements are dictionaries whose elements are lists whose elements are lists.

```
>>> myData = [{'IV6 -> I': {'[0, 4, 7] -> [-1, -2, 0]': [[15, 5], [16, 83]]}}, {}]
```

Thus to query all the locations of some particular voice leading, one writes:

```
>>> print myData[0]['IV6 -> I']['[0, 4, 7] -> [-1, -2, 0]']
    [[15, 5], [16, 83]]
```

The first parameter is a mode index (0 for major, 1 for minor), the second gives the chord progression we are interested in (here IV6 -> I), the third specifies an upper-voice voice-leading pattern (here, we are looking for a complete major triad in the upper voices, with the root moving down by semitone and the third moving down by two semitones, with the fifth held constant). The output is a list of lists, with the first value in each sublist being the chorale number, and the second value being the measure number. For the newcomer this sort of recursion can take a little bit of getting used to. Happily, Python

can work with gigantic data structures so that one can easily store—for example—the locations of all the voice leadings found in the Bach chorales.

In Python, one typically manipulates objects by calling their *methods,* which are represented by appending the method name to the object name, separated by a period. To add the number 5 to the end of myList, for example, one writes:

```
>>> myList.append(5)
```

Here, we call the "append" method of the object "myList," instructing it to add 5 to the end of the current list. Python has a large number of built-in methods for manipulating its data-types, making it particularly easy to work with strings, lists, and dictionaries.

It is worth noting that Python programmers tend to reserve capitalized names for *classes* (roughly: data structures combined with methods that can be applied to the data) rather than *methods* (roughly: executable subroutines) or *modules* (collections of methods and classes). This can confuse the uninitiated. For instance, to find the interval between two notes, we can write:

```
>>> a = pitch.Pitch('C4')
>>> b = pitch.Pitch('F4')
>>> i1 = interval.notesToInterval(a, b)
>>> print i1.semitones
5
```

The seemingly redundant "pitch.Pitch" is used to create instances of the class *Pitch* (capitalized, indicating a data structure representing an actual pitch) which is defined in music21's "pitch" submodule (uncapitalized; a collection of methods and classes). We then pass these two Pitches to the *method* "notesToInterval"; this returns an instance of the class "Interval," representing the interval from *a* to *b*. (Here the prefix "interval" labels the music21 submodule containing the "notesToInterval" routine; note that the label "notesToInterval" leaves the first letter uncapitalized, indicating that it is a method.) Alternatively, we could create an Interval object directly by writing:

```
>>> i1 = interval.Interval('P4')     # create an Interval representing a perfect fourth
```

Again, the redundancy indicates that the "Interval" class (capitalized) is found within the "interval" submodule (uncapitalized).

[SIDEBAR 3]

For instance, to get the first note of the top voice of Monteverdi's first Book 3 madrigal, one writes:

```
>>> myScore = corpus.parse('madrigal.3.1.mxl')  # read the score into music21
>>> print myScore[14][1][37]  # print element 37 of element 1 of element 14
```

(Note that the scores for books 3 and 4 of the madrigals are included with the program, along with a sizeable corpus of other scores.) Here, the top hierarchical layer, referenced by the first number in brackets (i.e. myScore[*x*]), contains the individual Parts (substreams), as well as a number of text boxes, some metadata, and other assorted information. The second hierarchical layer, referenced by the second number in brackets (i.e. myScore[14][*y*], contains the measures (substreams) in a single Part along with its instrument definition. Finally, the third hierarchical layer (myScore[14][1][*z*]) contains not just the actual notes in a particular method but also clefs, key signatures, measure-based text expressions, etc.

Of course, in this case, one could get the first note in the first measure of the first part by writing:

```
>>> print myScore.parts[0].measure(1).notes[0]
```

But using measure numbers in this way is quite risky, due to the problems created by unnumbered pickup measures, mid-measure repeats, second endings, and so forth. See the main text for more on this.

[SIDEBAR 4]

Here is the music21 code I wrote to find scales containing a given set of pitches; it is substantially faster than music21's built-in version. The code has two subroutines. The first (scale_from_letternames_init) allows the user to input the scales that she would like to search for; it converts a list of pitches into a list of *intervals from a given note to the tonics of scales containing it*. (Intervals, following music21 convention, are represented by strings in a natural way, with 'P1' representing the perfect unison, 'm2' the minor second, 'M2' the major second, and so on.) Thus the major scale becomes ['P1', 'm2', 'm3', 'P4', 'P5', 'm6', 'm7'] since the note C is contained within C major, D*f major, E*f major, F major, etc. These interval lists are stored in the global variable "storedScales," a list of lists of strings representing interval names.

The second routine (scale_from_letternames) takes an incoming list of pitches and uses the global "storedScales" variable to quickly compute a list of scales containing all of those pitches. This is a simple matter of transposing each note by each of the stored intervals and computing the intersection of the result. For this I use Python's "Set" data type.

```
from music21 import *
majInts = ['P1', 'm2', 'm3', 'P4', 'P5', 'm6', 'm7']          # intervals to tonics of scales containing a given note
melMinInts = ['P1', 'm2', 'm3', 'P4', 'P5', 'M6', 'm7']    # ... aka inversions of the scale around its tonic
harmMinInts = ['P1', 'm2', 'M3', 'P4', 'P5', 'M6', 'm7']

storedScales = [majInts, melMinInts, harmMinInts]    # global variable for the scales we are considering

def scale_from_letternames_init(newScale, reset=False):  # converts a list of pitches into a list of intervals
  global storedScales                                    # inverted around tonic note, for use in scale finding
  if reset:
    storedScales = []
  if newScale:
    tempScale = ['P1']
    for x in newScale[1:]:
          octaveComplement  = interval.notesToInterval(newScale[0], x).complement.name
          tempScale.append(octaveComplement)
    storedScales.append(tempScale)

def scale_from_letternames(myPitches):
  global storedScales
  results = []
  for myScale in storedScales:
    acceptableScales = set([myPitches[0].transpose(x).name for x in myScale])
    for i in range(1, len(myPitches)):
      newTonics = [myPitches[i].transpose(x).name for x in myScale]
      acceptableScales = acceptableScales & set(newTonics)
    results.append(acceptableScales)
  return results
```