LilyLib: A mid-level musical language

T.J.H. Morgan

10/4/2020

Table of Contents

[Visual tl;dr 3](#_Toc53415344)

[Introduction 4](#_Toc53415345)

[What is LilyLib? 4](#_Toc53415346)

[Why use LilyLib? 4](#_Toc53415347)

[How to run LilyLib 5](#_Toc53415348)

[What next? 5](#_Toc53415349)

[LilyLib core concepts 6](#_Toc53415350)

[Pieces and Notes 6](#_Toc53415351)

[Chords 11](#_Toc53415352)

[Scales 12](#_Toc53415353)

[Arpeggios, etc. 15](#_Toc53415354)

[Changing keys 17](#_Toc53415355)

[Other piece functions 18](#_Toc53415356)

[Slicing music 19](#_Toc53415357)

[Demo pieces 21](#_Toc53415358)

[Prelude in C 21](#_Toc53415359)

# Visual tl;dr

Speech =>

“Starting from the E above middle C, and in E Major. First the right hand plays an ascending 1 octave scale while the left hand plays a descending 1 octave arpeggio. Then the right hand plays a descending one octave dominant 7th, while the left hand plays a dominant 7th chord. Finally, both hands play a major chord.”

=> LilyLib =>

**class** **TLDR**(Piece):

**def** details(self):

self.title = "tl;dr demo"

self.key = 'E Major'

**def** write\_score(self):

self.score["treble"] =

scale('e`', 8, 8) +

dominant7('d``', -4, 4) +

[chord(arpeggio('e`', 4), 1)]

self.score["bass"] =

arpeggio('e`', -4, 4) +

[chord(dominant7('e', 5), 1)] +

[chord(arpeggio('e,', 4), 1)]

=> LilyPond =>

**\header** **{**

title = "tl;dr demo"

**}**

**\score** **{ <<**

**<<** **\new** **Staff** **{**

**\clef** treble **\key** e **\major** **\time** 4/4

e'8 fs'8 gs'8 a'8 b'8 cs''8 ds''8 e''8 d''4 b'4 gs'4 e'4 <e' gs' b' e''>1**} >>**

**<<** **\new** **Staff** **{**

**\clef** bass **\key** e **\major** **\time** 4/4

e'4 b4 gs4 e4 <e gs b d' e'>1 <e, gs, b, e>1**} >>**

**>> }**

=> Sheet music

A screenshot of a cell phone

Description automatically generated

# Introduction

## What is LilyLib?

When musicians talk about a piece of music they don’t do so note-by-note, rather they talk about higher-level structures; scales, arpeggios, harmonies, motifs, key changes, and so on. Nonetheless sheet music adopts the note-by-note approach, as does Lilypond—a computer language for creating sheet music. Here’s a basic example illustrating this difference:

“Play a descending scale in C minor, from middle C, with a swing rhythm.”

Diagram, schematic

Description automatically generated

But what if we could create sheet music directly from the higher-level structures musicians talk about? This is where LilyLib comes in: LilyLib is a python library that allows the user to describe a piece of music in terms of the concepts that are familiar to musicians. LilyLib automatically compiles into Lilypond, which can itself be compiled into sheet music. Here’s the LilyLib for the scale above:

self.score["bass"] = self.scale('c`', 'c', ['8.', 16], key="C Minor")

Not *quite* as familiar as spoken language, but note that it does not list the notes of the scale. Instead it calls a function called scale, providing it with the key, the start and stop points, and a list of rhythms. LilyLib does the rest: when this file is executed it outputs Lilypond which can then be converted into the pdf shown above.

## Why use LilyLib?

*LilyLib is scalable*

Many pieces repeat certain motifs or patterns. LilyLib lets the user extract these as functions and repeatedly apply them as necessary. LilyLib even understands concepts such as transposition, so if a motif is repeated but in a different key, that can all be handled automatically. This means that even long pieces, consisting of thousands of notes, can often be expressed as a relatively short amount of code.

*LilyLib encourages musical analysis*

To most effectively take advantage of LilyLib, the user should decompose a piece of music into its motifs, key signatures and overall structure. In doing so the users gains insight into how the music works at a higher level than is required when reading sheet music.

*LilyLib allows high level manipulation of music*

The whole point of LilyLib is to describe music at a higher level, “refactoring” it in programming terms. Once this is done, unusual manipulations of sheet music that would take a lot of effort with traditional methods take just a few minutes, or even seconds. For instance, transposing an entire piece into a different key, changing the rhythm of a motif, switching the hands, or reducing a piece to a list of its unique motifs. We’ll see some of these later.

## How to run LilyLib

Executing a LilyLib file once it is created take just a few seconds. First you will need to have installed python 3 and a Lilypond compiler (e.g. Frescobaldi). After that you should download LilyLib from github.

Any files you wish to run need to be inside the LilyLib directory, and you can run them with the terminal command “python3 <file\_name.py>”. This will spit out the corresponding Lilypond into the terminal which can be pasted into your Lilypond compiler to produce a pdf.

## What next?

The rest of this guide walks you through the key concepts of LilyLib, illustrating each with some toy examples. After that there are a couple of examples of full pieces that had been converted to LilyLib. All the demos are included in the github repo, as is all the source code, so you should feel free to open up and explore the files themselves. The hope is that by the end of this guide the reader should have a vague sense of how to write in LilyLib should they so desire.

# LilyLib core concepts

## Pieces and Notes

The two classes even the most basic LilyLib composition will interact with are Piece and Note. To illustrate here’s the sheet music and full LilyLib for creating a piece of music that is simply both hands playing Middle C:

A picture containing text

Description automatically generated

**from** **piece** **import** Piece

**class** **MiddleC**(Piece):

**def** details(self):

self.title = "Middle C"

**def** write\_score(self):

self.score["treble"] = self.notes("c`", 1)

self.score["bass"] = self.notes("c`", 1)

**if** \_\_name\_\_ == "\_\_main\_\_":

MiddleC()

Let’s go through this line by line. First we import the base class Piece from LilyLib.

**from** **piece** **import** Piece

This is critical because otherwise python doesn’t know what a Piece is. The next line is more python syntax that declares out intention to make a new Piece:

**class** **MiddleC**(Piece):

We’ve imaginatively called our Piece **MiddleC** and we tag “(Piece)” on the end to let python know that **MiddleC** is a kind of Piece. If this is confusing to you, you should take an intro to python course and then everything will become clear.

Our new piece contains two functions: details and write\_score. These contain the meta-data and the musical content of the piece respectively and all pieces must have them. In the details function here we are titling our piece “Middle C”, this is the text that will appear at the top of the pdf after compilation, but we can set other things too. To see what, open piece.py and look at the \_\_init\_\_ function right at the top:

**class** **Piece**:

**def** \_\_init\_\_(self):

self.title = ""

self.subtitle = ""

self.composer = ""

self.opus = ""

self.staves = [Treble(), Bass()]

self.tempo = "4/4"

self.key = "C Major"

self.score = {}

self.tonespace = ToneSpace()

self.create\_key\_dictionary()

self.details()

self.set\_key(self.key)

**print**(self)

From top to bottom this function sets default values for things like the title, subtitle, composer and so on, before later calling self.details(). By writing our own details function and having it change these values we can overwrite things like the title, tempo or key signature, and so on. Here’s an example of how changing the details function changes the resulting pdf:

**class** **MiddleC**(Piece):

**def** details(self):

self.title = "Middle C"

self.subtitle = "A shockingly avant-garde production"

self.composer = "Lily Lib"

self.opus = "Op.3 No.2"

self.tempo = "7/8"

self.key = "FS Major"

**def** write\_score(self):

self.score["treble"] = self.notes("c`", 1)

self.score["bass"] = self.notes("c`", 1)

Text

Description automatically generated

The write\_score function does most of the work though, and for complex pieces it will get quite complex too. But, whatever the piece is, the requirement is that it modifies the score of the Piece (referred to as self.score) adding the contents of the staves. Here our piece has the default staves: one treble, one bass. We refer to these staves by name; self.score["treble"] and self.score["bass"].

In this demo, we write the score using the function self.notes. This is a function of the base class Piece (so you can look it up in piece.py if you want) and, as the name suggests, it makes Notes. Notes are another class in LilyLib and you can look them up in models.py. The Note class is very basic, it has a Tone (the root frequency of the note, or its vertical placement on a staff) and a duration, it can also have ornamentation (including dynamics, trills, mordents, etc.), but that’s it. Notes don’t *do* anything, but when they print out they return a Lilypond description of themselves.

***Digression: Notes vs. Tones***

*If you look at the code for the Note class, you’ll see that the Note is not the most basic element: Tone is itself a class, though not one that users will interact with directly very much. A Tone can be thought of as a durationless Note (it’s ornamentation-less too). As such it’s a pretty abstract concept. When LilyLib executes a Piece, early on it makes a set of Tones corresponding to every possible sound an instrument can make, and it them stores these in a “tonespace”, then, when Notes are created, LilyLib draws on Tones from this pool. In this way, while there can be several different Notes using the same Tone, there is only a single Tone for each possible sonic frequency an instrument can produce. Users should not attempt to make new (duplicate) Tones, and LilyLib will try to stop you doing this.*

The self.notes function takes a broadly note-by-note approach, but it has some neat tricks. It takes three arguments: the tones of the notes, the duration of the notes and the ornamentation of the notes. All three arguments can be lists of multiple values, or a string with multiple values separated by spaces, and the function assumes you want as many notes as the length of the longest argument, with the shorter arguments being cycled to get you there. Here’s a few examples:

Multiple tones that share a single duration:

self.score["treble"] = self.notes("c` c` c` c`", 1)

self.score["bass"] = self.notes("c` c` c` c`", 1)

Table

Description automatically generated

A single tone, but four durations and 2 ornaments

self.score["treble"] = self.notes('c`', '4. 8 4 4', "~ ")

self.score["bass"] = self.notes('c`', '4. 4 8 4', " ~")

Diagram

Description automatically generated

Note that the whitespace before or after the ~ symbol (which is Lilypond notation for a tie) is enough to tell LilyLib to apply the tie to every other note. Lastly, because the notes function returns a list of Note objects, and because python lets you multiply lists to duplicate their contents, we can also multiply the results of the notes function to continue a motif. Here’s an example:

self.score["treble"] = self.notes('c` c` f` e`', '4 8 4. 4', "~ ") \* 2

self.score["bass"] = self.notes('c g g c', '4. 8 8 4.', " ~") \* 2

A close up of a antenna

Description automatically generated

## Chords

LilyLib also includes the class Chord. Chords are much like Notes, except that they are associated with multiple Tones instead of just one. However, just like a Note, they have a duration and an optional ornamentation. The easiest way to make a chord is with the self.chord function in the Piece class. Here’s an example in the C Major Chord demo:

**class** **CMajorChord**(Piece):

**def** details(self):

self.title = "C Major Chord"

**def** write\_score(self):

self.score["treble"] = self.chord("c` e` g` c``", 1)

self.score["bass"] = self.chord("c, c", 1)

Which compiles to:

A picture containing text

Description automatically generated

Note that the chord function behaves just like the notes function: it takes the notes that comprise the chord as its first argument, and then the duration of the chord. Here we have specified the notes as a string, separated by spaces, but, as with notes, you can also pass a list of tones.

## Scales

LilyLib is not just a nice wrapper for creating notes and chords one-by-one. Its real utility comes with concepts that relate to patterns across multiple notes, and we’ll start with scales. The demo “c\_major\_modal\_scales.py” shows a whole bunch of different ways scales can be made. At the most horrible end, is building the scale by calling the notes function for each individual note:

"treble": [self.notes("c`", 8), self.notes("d`", 8), self.notes("e`", 8), self.notes("f`", 8), self.notes("g`", 8), self.notes("a`", 8), self.notes("b`", 8), self.notes("c``", 8)],

We’ve already mentioned how this can be reduced to a single call though:

"treble": self.notes('d` e` f` g` a` b` c`` d``', 8)

But the scale function is even nicer: you specify a start-tone, and then either a stop-tone or a length (where length is negative the scale is descending), followed by a duration (as per the notes function). Here the treble clef passes a stop-tone, while the bass clef passes a length:

"treble": self.scale('e`', 'e``', 8),

"bass": self.scale('e', 8, 8)

But we can draw on the power of python to do something even fancier. Lets say we want to do the following:

“Play an ascending C Major scale with both hands one octave apart, then do it again but starting from D (staying in C Major though), then from E, and so on, until you start from C again but a full octave above where you began.”

This whole operation can be compressed to a few lines of LilyLib as follows:

looped = {"treble": []}

**for** start **in** self.scale('c`', 'c``', 8):

looped["treble"] += self.scale(start, 8, 8)

looped["bass"] = self.transpose(looped["treble"], -1)

Line-by-line here’s what this does:

1. It makes a dictionary called “looped” with a single key (“treble”) whose value is an empty list.
2. It then makes a one octave scale from middle C, and goes through every note in that scale
3. For each note, it makes a new scale, 8 notes long, starting from that note, and adds it to looped[“treble”]
4. Lastly it uses the transpose function to state that the bass clef is the same as the treble clef, just transposed down one octave (-1).

This is what the sheet music looks like:

Diagram

Description automatically generated

By this point we are getting serious time savings, and you can probably imagine that LilyLib allows Hanon-esque exercises to be written very quickly.

Here’s another example, even wilder, than uses the “step” argument which produces scales that take larger steps (step=2 means play every other note), and, the key argument to change the keys of the scales (if you don’t specify a key, it defaults to the overall key of the piece). Here’s the code:

start\_notes = self.scale('c```', 'c``')

smart = {

"treble": [self.scale(start, -8, 8, step=2, key=start.letter +

" major") **for** start **in** start\_notes],

"bass": [self.scale(self.transpose(start, -1), -8, 8,

key=start.letter + " major") **for** start **in** start\_notes]

}

This code is not very human-readable, especially if you are new to python, so it would take a while to read this *instead* of sheet music. LilyLib doesn’t care though and converts it to Lilypond (and then to pdf) without difficulty. Here’s the sheet music:

Diagram, schematic

Description automatically generated

For comparison, here’s the Lilypond code for this exercise:

**\score** **{ <<**

**<<** **\new** **Staff** **{**

**\clef** treble

**\key** c **\major**

**\time** 4/4

c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8 d'8 e'8 f'8 g'8 a'8 b'8 c''8 d''8 e'8 f'8 g'8 a'8 b'8 c''8 d''8 e''8 f'8 g'8 a'8 b'8 c''8 d''8 e''8 f''8 g'8 a'8 b'8 c''8 d''8 e''8 f''8 g''8 a'8 b'8 c''8 d''8 e''8 f''8 g''8 a''8 b'8 c''8 d''8 e''8 f''8 g''8 a''8 b''8 c''8 d''8 e''8 f''8 g''8 a''8 b''8 c'''8 c'''8 a''8 f''8 d''8 b'8 g'8 e'8 c'8 b''8 gs''8 e''8 cs''8 as'8 fs'8 ds'8 b8 a''8 fs''8 d''8 b'8 gs'8 e'8 cs'8 a8 g''8 e''8 c''8 a'8 fs'8 d'8 b8 g8 f''8 d''8 bf'8 g'8 e'8 c'8 a8 f8 e''8 cs''8 a'8 fs'8 ds'8 b8 gs8 e8 d''8 b'8 g'8 e'8 cs'8 a8 fs8 d8 c''8 a'8 f'8 d'8 b8 g8 e8 c8**} >>**

**<<** **\new** **Staff** **{**

**\clef** bass

**\key** c **\major**

**\time** 4/4

c8 d8 e8 f8 g8 a8 b8 c'8 d8 e8 f8 g8 a8 b8 c'8 d'8 e8 f8 g8 a8 b8 c'8 d'8 e'8 f8 g8 a8 b8 c'8 d'8 e'8 f'8 g8 a8 b8 c'8 d'8 e'8 f'8 g'8 a8 b8 c'8 d'8 e'8 f'8 g'8 a'8 b8 c'8 d'8 e'8 f'8 g'8 a'8 b'8 c'8 d'8 e'8 f'8 g'8 a'8 b'8 c''8 c''8 b'8 a'8 g'8 f'8 e'8 d'8 c'8 b'8 as'8 gs'8 fs'8 e'8 ds'8 cs'8 b8 a'8 gs'8 fs'8 e'8 d'8 cs'8 b8 a8 g'8 fs'8 e'8 d'8 c'8 b8 a8 g8 f'8 e'8 d'8 c'8 bf8 a8 g8 f8 e'8 ds'8 cs'8 b8 a8 gs8 fs8 e8 d'8 cs'8 b8 a8 g8 fs8 e8 d8 c'8 b8 a8 g8 f8 e8 d8 c8**} >>**

**>> }**

## Arpeggios, etc.

LilyLib isn’t limited to scales though, and it also provides the following functions:

arpeggio – returns arpeggios.

arpeggio7 – returns arpeggios but including the 7th too.

dominant7 – returns dominant 7ths.

diminished7 – returns diminished 7ths.

chromatic – returns a chromatic scale.

scale\_subset – returns a customizable subset of the overall scale.

These behave very much like scale. You specify a starting note, then either a stop note or a length, and you can additionally specify the duration, ornamentation, key and step size. Note that whatever note you specify as the start or stop points must exist in the relevant series, otherwise you’ll get an error. So, asking for a scale in C Major, but starting on F sharp just won’t work.

The arpeggios demo file gives you some examples of how the arpeggio function can be used. Here’s a simple one:

"treble": self.arpeggio('f`', 4, 8, key="F Major"),

"bass": self.arpeggio('f', 4, 8, key="F Major")

A picture containing antenna

Description automatically generated

But here’s something a bit fancier:

starts = self.arpeggio('c`', 'c``')

'treble': [[self.arpeggio(start, self.transpose(start, 1), 16, step=step) **for** step **in** [3, 3, 1]] **for** start **in** starts],

'bass': [[self.arpeggio(self.transpose(start, -1), start, 16, step=step) **for** step **in** [1, 3, 3]] **for** start **in** starts]

This nested list comprehension is very pythonic, and not very human-readable, but it produces this passage which is vaguely reminiscent of Beethoven’s 3rd piano sonata:

A picture containing antenna, guitar

Description automatically generated

Note how the step argument is used to switch between octaves and full arpeggios.

## Changing keys

So far, all the demos have set the key once at the start of the piece (in the details function, it defaults to C Major), but many pieces change key midway through. Here’s how to do it:

**def** write\_score(self):

self.score["treble"] = self.chromatic('c`', 'c``', [16]\*12 + [4])

self.set\_key("f major")

self.score["treble"] +=

self.key\_signature + self.chromatic('f`', 'f``', [16]\*12 + [4])

self.score["treble"][-1].dur = 4

self.score["bass"] = self.transpose(self.score["treble"], -1)

Diagram, schematic

Description automatically generated

This code first creates a chromatic scale from C to C using the chromatic function. It then uses the function self.set\_key to change to F Major and then writes a second chromatic scale from F to F. The set\_key function accepts strings and all “real” keys are permitted (i.e. excluding impossible or theoretical keys, like G-sharp Major). It’s also case insensitive, just note that it expects Lilypond notation for the letter, so “BF Major”, not “B Flat Major”. Lastly, LilyLib distinguishes between natural and harmonic minor keys, so you can use “D Minor” or “D Minor harmonic” (which also be written as “D harmonic minor”). With harmonic minor keys the 7th is automatically sharpened by functions like scale.

While the set\_key function changes the key, it does *not* print the new key signature to the sheet music, this lets you “invisibly” change the key. To additionally print the key signature, you must add self.key\_signature to the score, as is done above. Note that harmonic and natural minors print the same key signature, even though they produce different 7ths.

For an even fancier version of key changes, that cycles through every single key, see demo\_keys.py.

## Other piece functions

Here’s some other functions that the Piece class offers that we’ll see used later.

tones – a function that converts things (like strings) into Tone objects

rests –like the notes function, but returns rests instead; just specify the durations.

harmonize – you give it a list of notes and intervals, and it returns a list of chords wherein the notes have been harmonized at those intervals.

triplets – makes a list of notes triplets

voices – give it multiple lists of notes/chords and it returns them as a single passage composed of multiple voices

repeat – wraps a series of notes/chords in pretty repeat bars. You can specify how many times it should be repeated and (if more than 2) it will print the number over the bar lines.

annotation – insert it into the score to have a text annotation appear in the sheet music

I, II, III, … VI, VII – returns the Major key at the specified interval relative to the current key. For instance, for a piece in C Major, III returns E Major, so to make this switch you can do:

self.set\_key(self.III)

i, ii, iii, … vi, vii – returns the Minor key at the specified interval relative to the current key

ih, iih, iiih, … vih, viih – returns the Minor harmonic key at the specified interval relative to the current key

Ic, IIc, IIIc, … VIc, VIIc – returns the cis\* key at the specified interval relative to the current key

It, IIt, IIIt, … VIt, VIIt – returns the trans\* key at the specified interval relative to the current key

\* a cis key is in the same mode as the current key, whereas trans keys are in the opposite mode. So cIII of C Major is E Major, whereas tIII is E Minor.

## Slicing music

Most of the functions described above return Lists of Notes and Chords. Moreover, each staff in the score (e.g. self.score[‘treble’]) is just a List too. Lists are a python object, and python uses square-brackets to select subsets or single items within lists. However, python also counts from 0, which creates some oddities. So, if you wanted the first note from the bass clef, you can do this:

first\_note = self.score['bass'][0]

Note that the “first” note is actually the 0th as far as python is concerned. You can take a subset too. Say you want the 5th, 6th, 7th and 8th notes of the bass clef:

four\_notes = self.score['bass'][4:8]

Here we start on the 4th note (which is conventional parlance is the 5th) and we go up to the 8th (a.k.a. the 9th) but that’s not included because python slices are inclusive for the first index, but not for the second. Altogether, this system does not match how most people think about indexes. There’s also no easy way to, for instance, take the 1st, 3rd, 6th and 7th notes. To this end LilyLib offers its own slicing functions:

Get the first note (counts from 1, not 0):

first\_note = select(self.score['bass'], 1)

Get the first and 4th notes:

two\_notes = select(self.score['bass'], 1, 4)

Note that the order of the indexes does not matter for this function. So the following is the same as the above:

the\_same\_two\_notes = select(self.score['bass'], 4, 1)

select just returns all notes whose indexes are in function call, so it ignores repeated indexes too:

still\_the\_same\_two\_notes = select(self.score['bass'], 4, 1, 1, 1, 4)

For a more powerful function that specifies order and allows repeats, see pattern below.

Get all notes from the 5th to the 8th (both indexes are inclusive):

four\_notes = subset(self.score['bass'], 5, 8)

Get the 8th to the 5th notes, *in reverse order*:

reverse\_notes = subset(self.score['bass'], 8, 5)

Remove the 4th and 7th notes:

remaining\_notes = remove(self.score['bass'], 4, 7)

Note that remove is basically the inverse of select, so it returns a list of all items whose index was not passed to the remove function, keeping order the same and ignoring repeats.

Create a custom reordering of any subset of notes (including repeated notes), here it’s the 1st, 5th, 6th, 1st again, 1st *again*, and lastly the 6th again:

new\_notes = pattern(self.score['bass'], 1, 5, 6, 1, 1, 6)

All these slicing functions are in util.py.

# Demo pieces

## Prelude in C

**Simple:**

As a first stab at writing actual music, let’s begin with Bach’s prelude in C. to start we can simply create the class and set the details:

**class** **PreludeInC**(Piece):

**def** details(self):

self.title = "Prelude in C"

self.composer = "J. S. Bach"

self.opus = "BVW 846"

Anyone familiar with this piece knows that almost the entire piece is built around a repeating motif. So the first thing we’ll do is write a function that creates this motif, but without knowing what the specific notes are. We can create this as a subfunction within write\_score:

**def** write\_score(self):

**def** motif(c):

tones = self.tones(c)

self.score["bass"] += 2 \* voices(

rests(16) + notes(select(tones, 2), ['8.', 4], "~ "),

notes(select(tones, 1), 2)

)

self.score["treble"] += 2 \* (

rests(8) + notes(pattern(tones, 2\*[3, 4, 5]), 16)

)

Let’s walk through it. So the motif function accepts an argument called c (for “chord”) and immediately converts it into a list of tones. It then adds to (+=) the bass and treble clefs. The bass clef is split into two voices. The first voice is a semiquaver rest, followed by the 2nd tone of the chord repeated twice (but linked with a tie). The second voice is just the first tone of the chord, held for a minim. Note that voices start from high to low, this is Lilypond convention, other orderings produce weird looking sheet music. The treble clef is a quaver rest followed by the 3rd, 4th, 5th tones of the chord, all semiquavers, played twice. Note that both staves have the pattern repeated (“2 \*” at the start).

***Digression: Python and “self”***

*You’ve probably noticed that lots of LilyLib code calls functions with the word “self” attached to the front. This is python behavior, and it indicates whether the function is a part of the piece class (in which case “self” is appended) or not. However, it doesn’t help readability, and you may also have noticed that the motif function above does not mention self, calling, for instance, the notes and rests functions directly. This is done with a pythonic trick, in which the functions are relabeled before the motif function is created:*

notes, rests, voices = self.notes, self.rests, self.voices

With this function done, we then just need to list the chords. The simplest way (demo\_prelude\_simple.py) is to name all the notes:

bar = [''] \* 40

bar[1] = 'c` e` g` c`` e``'

bar[2] = 'c` d` g` c`` e``'

bar[3] = 'b d` g` d`` f``'

bar[4] = 'c` e` g` c`` e``'

First we make a list of empty strings called “bar”, we then start listing the bars one-by-one (we are leaving bar[0] empty, because sheet music counts bars from 1 and this way we can easily look up any bar in the piece).

Once all chords are written, we pass each chord to the motif:

**for** c **in** bar:

**if** c:

motif(c)

The “if c:” is necessary because bar[0] is empty and we want to skip any empty bars. After that we manually write out the final few bars that deviate from the motif:

self.score['treble'] += rests(8) + notes('f a c` f` c` a c` a f a f d f d', 16)

self.score['treble'] += rests(8) + notes('g` b` d` f` d` b` d` b` g` b` d` f` e` d`', 16)

self.score["bass"] += 2 \* voices(rests(16) + notes('b,', ['8.', 4, 2], "~ ~ "), notes('c,', 1))

self.score['treble'] += [chord('e` g` c`', 1)]

self.score['bass'] += [chord('c, c', 1)]

And we are good to go; we can execute the file (demo\_prelude\_in\_c\_simple.py) and compile it into a full pdf.

**Intermediate:**

The method of writing out each chord note-by-note used above is not great. It ignores that the chords can be understood at a higher level, the first, for instance, is a C Major arpeggio. To that end, demo\_prelude\_in\_c.py shows an alternative way to write the chords. Here’s an example:

bar[1] = arpeggio('c`', 'e``')

bar[2] = remove(dominant7('c`', 'f``', key='D Minor'), 3, 5)

bar[3] = remove(dominant7('b', 'f``', key='G Major'), 3, 5)

bar[4] = bar[1]

bar[5] = remove(arpeggio('c`', 'a``', key='A Minor'), 4)

bar[6] = tones('c`') + arpeggio('d`', 'd``', key='D Major')

bar[7] = transpose(bar[5], -1, 'scale')

bar[8] = tones('b') + arpeggio('c`', 'c``')

Some of these are easy to read: bar[1] is a simple arpeggio; bar[4] is just bar[1]. But often the chords don’t include all notes in their range, so bar[2] and bar[3] are both dominant 7ths, in different keys, and both skip the 3rd and 5th notes in their range. Details like this make the code harder to read, and sometimes you wonder if it’s easier to specify which notes are removed versus which are kept in. But, overall, the process of figuring out what each chord is pays off.

**Advanced:**

OK, now for some cool stuff. Let’s say you are confident you can remember the motif itself, and so all you want is a piece of sheet music showing the chords, and you can handle the rest. Well, we can do this very easily by modifying the motif function:

**def** motif(c):

**if** self.summary:

bass = [chord(subset(tones, 1, 2), 4)]

treble = [chord(subset(tones, 3, 5), 4)]

**else**:

bass = 2 \* voices(

rests(16) + notes(select(tones, 2), ['8.', 4], "~ "),

notes(select(tones, 1), 2)

)

treble = 2 \* (

rests(8) + notes(pattern(tones, 2\*[3, 4, 5]), 16)

)

self.score["bass"] += bass

self.score["treble"] += treble

Here the motif function does one of two things. If self.summary is true it produces single chords, otherwise it prints the motif. The summary variable is set in the details function:

**def** details(self):

self.title = "Prelude in C"

self.composer = "J. S. Bach"

self.opus = "BVW 846"

self.summary = True

Here’s what this produces:

Chart, box and whisker chart

Description automatically generated

OK, one last thing: We’ve already been through the process of figuring what all these chords are, but this gets a little lost in the sheet music, so let’s add the chord names to the music too. First, let’s give all the chords a name:

bar[1] = (arpeggio('c`', 'e``'), 'I')

bar[2] = (remove(dominant7('c`', 'f``', key='D Minor'), 3, 5), 'ii D7')

bar[3] = (remove(dominant7('b', 'f``', key='G Major'), 3, 5), 'V D7')

bar[4] = bar[1]

Here, each bar is a “tuple” (a python object of two things put together inside parentheses). The first item in each bar is the chord, the second is the chord’s name that we’ll see in the sheet music. We can then modify the motif function to annotate the sheet music:

**def** motif(c):

tones = c[0]

**if** self.summary:

bass = [chord(subset(tones, 1, 2), 4)]

treble = [chord(subset(tones, 3, 5), 4)]

**else**:

bass = 2 \* voices(

rests(16) + notes(select(tones, 2), ['8.', 4], "~ "),

notes(select(tones, 1), 2)

)

treble = 2 \* (

rests(8) + notes(pattern(tones, 2\*[3, 4, 5]), 16)

)

**if** self.annotate:

self.name(treble, c[1])

self.score["bass"] += bass

self.score["treble"] += treble

It’s the same as before, but now it additionally checks the variable self.name (defined in the details function) and if it’s true it uses the self.name function to add text to the score. With both annotations and summary set to true, here’s the full sheet music:

Shape

Description automatically generated

## Mad Rush

Mad Rush by Philip Glass is a great example of LilyLib’s ability to compress a piece because it is highly repetitive; the piece is about 14 minutes long, and runs to about 20 pages of sheet music, but we can eventually get it down to just a couple of pages.

The piece has four main parts that are strung together in various ways: (A) the quiet part, where a chord is played in simple time in the left hand and compound time in the right, (B) the fast, loud part, where rapid arpeggios are played in both hands, alternating between 12/8 and 14/8, (C) a hybrid of the two where the left hand plays A and the right hand plays a variation on B, (D) the “outro”, similar to A, but the right hand does something else. Traditional sheet music tries to accommodate this structure with a bunch of codas and annotations to jump to other parts of the sheet music, but we can do better.

The demo code builds all four sections in much the same way. So we’ll go through A as an illustration. First we define some chords that are used in the section:

aI = self.arpeggio('f', 6)

aiii = self.arpeggio('e', 6, key='A Minor')

aiii7 = pattern(self.arpeggio7('g', 7, key='A Minor'), [1, 2, 3, 5, 6, 7])

aii = self.arpeggio('g', 6, key='G Minor')

aii7 = self.tones('f') + subset(aii, 2, 6)

Here the chords are named according to their relationship to the root key, with the prefix ‘a’ to indicate they are used in section A. So, ‘aiii7’ is a clue that this chord is in the minor third (A Minor), including the 7th.

There are then three functions to build segments of music for section A. The first two, ‘doublet\_bar’ and ‘triplet\_bar’, take a pair of notes and return a bar of music in which those notes are played as alternating quavers in either simple time (doublet\_bar) or compound time (triplet\_bar), you can also specify how many bars you want (including decimal numbers). The other function, ‘A\_motif’, builds 4-bar chunks. The function is passed a chord and it then, passes the relevant bits of the chord to doublet\_bar and triplet\_bar, it then assigns the output to the different staves and tweaks it where necessary. So here it builds the treble clef according to which chord it’s working with:

**if** chord == aI:

motif['treble'] = triplet\_bar(pattern(chord, [6, 5]), bars=bars)

**elif** chord == aii **and** 'low triplets' **in** tweaks:

motif['treble'] = triplet\_bar(pattern(chord, [5, 4]), bars=bars)

**else**:

motif['treble'] = triplet\_bar(pattern(chord, [6, 4]), bars=bars)

The only difference is which notes are used to create the triplets. Here’s an example of a tweak being applied:

**if** 'no treble' **in** tweaks:

motif['treble'] = self.rests(1) \* bars

So the opening of section A differs from the rest in having an empty treble clef. The tweak ‘no treble’ replaces the treble clef with rests. The output of this function is saved to a dictionary called ‘sections’. Here’s the simplest case, where the subsection ‘A2’ is created:

sections['A2'] = merge(A\_motif(aI, 2), A\_motif(aiii, 2))

So subsection 2 of section A is the main motif of section A applied to the root chord (aI) for 2 bars, and then to the minor 3rd (aiii) for 2 bars. The other subsections are a little more complex, but once they are all created they can be strung together to form the overall section:

A = ['A1', 'A2', 'A2', 'A3', 'A3', 'A4', 'A4', 'A5']

This process is then repeated for sections B, C and D, and then the overall piece structure can be defined:

structure = [A, A, 'A1', B, 'A1', C, 'A1', C, 'A1', B, A, A, 'A1', B, 'A1', D, D, 'A1']

Note how the subsection A1 gets used as a bridge between different sections a lot. The last bit is then to actually take the list of (sub)section names in the variable ‘structure’ as well as the actual description of each subsection in the ‘sections’ dictionary and write the score:

self.score["treble"] = []

self.score["bass"] = []

**for** section **in** flatten(structure):

self.score["treble"] += sections[section]['treble']

**if** 'B' **not** **in** section:

self.score['bass'] += self.voices(sections[section]['bass1'],

sections[section]['bass2'])

**else**:

self.score['bass'] += sections[section]['bass']

This goes through every subsection listed in ‘structure’ and then appends the relevant music from `sections` onto the end of `score`. Note that it’s a little fiddly because sections A, C and D have two voices in the bass clef, but section B doesn’t.

With this you can execute the file and get the sheet music. It compiles to about 20 pages in Lilypond, whereas the code prints to just 2 or 3.

Now that we have the music, there’s some nicer things we can do though. For one, how about we flag each subsection with its name, so we can read them as we go. Here’s how to do this:

**for** section **in** sections:

self.name(sections[section]['treble'], section)

A slightly counterintuitive mix of python and LilyLib, but this names the music as follows:

Diagram

Description automatically generated

But let’s do even more. Why not have a text list of the order of (sub)sections and just print the notes for each section once as a reference? The demo does this, using the variable ‘self.summary’ to switch between full and summary modes. The first change it to have the bit of code that adds to the score change it’s behavior when ‘summary’ is set to True. Specifically, in summary mode, rather that read the piece’s structure, it just reads from the list of all subsections:

self.score["treble"] = []

self.score["bass"] = []

**if** self.summary:

sections\_to\_print = sections

**else**:

sections\_to\_print = flatten(structure)

**for** section **in** sections\_to\_print:

self.score["treble"] += sections[section]['treble']

**if** 'B' **not** **in** section:

self.score['bass'] += self.voices(sections[section]['bass1'],

sections[section]['bass2'])

**else**:

self.score['bass'] += sections[section]['bass']

The next step is to create a text description of the structure that can be printed at the top of the page. You can do this with the ‘subtext’ function, which we haven’t seen before:

**def** subtext(self):

**if** self.summary:

**return**"""

**\\**markup {

**\\**column {

**\\**line {**\\**bold {Overall:} **\\**bold{A}, **\\**bold{A}, A1, **\\**bold{B}, A1, **\\**bold{C}, A1, **\\**bold{C}, A1, **\\**bold{B}, **\\**bold{A}, **\\**bold{A}, A1, **\\**bold{B}, A1, **\\**bold{D}, **\\**bold{D}, A1}

**\\**line { - }

**\\**line {**\\**bold {Section A:} A1, A2, A2, A3, A3, A4, A4, A5}

**\\**line {**\\**bold {Section B:} B1, B1, B2, B2, B2, B3, B3, B4}

**\\**line {**\\**bold {Section C:} C1, C1, C2, C2, C3, C3, C4}

**\\**line {**\\**bold {Section D:} D1, D1, D2, D2, D3, D3, D4}

**\\**line { - }

}

}"""

**else**:

**return** ""

This uses Lilyponds markup language, so you are forgiven for not understanding it straight away. But with this done, we get the following:

A picture containing text, application

Description automatically generated

With this the piece is reduced to three and a half pages of sheet music with a little text topper. It’s a really elegant way to see the whole piece at once.

**Advanced:**

One last feature. Because LilyLib understands key signatures it is in theory possible to ask it to rewrite a piece in any key. A basic way to do this is just to change the key signature, but this will often cause issues when specific notes are referenced. For instance, in the mad rush demo, when the chords are built, specific notes are named as the start and stop points of different chords:

aI = self.arpeggio('f', 6)

aiii = self.arpeggio('e', 6, key='A Minor')

So if we change the key to D Major, asking for an arpeggio starting on ‘f’ will cause LilyLib to break, because ‘f’ is not a part of D Major. Another issue is that, whatever we set the starting key to, the chord ‘aiii’ is hardcoded to be in A Minor, so that will ignore any key changes. Nonetheless, with some care it is possible to build the chords such that the piece can be written in any key. There are two key steps:

1. Define all notes, not by their name, but in reference to a "root note” (the root of the starting key is easiest).
2. Define all keys, not by their name, but in reference to the starting key.

Doing this, we get the following code:

aI = self.arpeggio(self.key.root, 6)

aiii = self.arpeggio(self.transpose(self.key.root, -1, "scale"), 6, self.tIII)

Note we use, ‘self.key.root’ instead of ‘f’, ‘self.transpose(self.key.root, -1, ‘scale’)’ instead of ‘e’ and ‘self.tIII’ instead of ‘A Minor’ (this is a trans key, see above for more information). Applying this throughout the piece takes about 15 minutes, but then we can change the key to whatever we want. As proof, here’s Mad Rush in… C Sharp Minor:

A picture containing table

Description automatically generated