Documentation for Pattern Composer

(This documentation is not complete and might not reflect all the functionality available as I'm continually updating the program)

Pattern Composer is a program for writing music in code that will be converted into a midi file. It centers around a chord progression and a timeline, and allows for creating patterns that respond to the chords and particular points in time during the song. The programming language used is Python.

Syntax

General structure

The output of the code is based on a structure, where you assign a list of lists to the reserved variable PARTS. Each list represents a part in a song, where the first entry is a chord progression and the second is a list of patterns. This is an example of a valid structure:

```
chorus = (('D#m', 4), ('G#7', 4))
verse = (('A#7', 4), ('A#', 4))

PARTS = [
        [chorus*2, [chord_stabs, drums, bass]],
        [verse*2, [chord_stabs,  drums, bass]]
]
```

chords_stabs, drums and bass are functions that represent patterns. chord_stabs will add notes to the first track, drums will add notes to the second track and bass to the third. Their index in the list determines which track they belong to.

Each part lasts for as long as the chord progression lasts.

Chords are represented as strings in the style of 'A#m', which you would typically see in other places. The program supports most chords, but inverted chords are not available yet ('C/G' for

example). The integer in the second entry of the tuple represents the duration of the chord in quarter notes.

Patterns

Consider the following:

```
def drums():
    if is_quarter_note:
        play(BASS_DRUM)
    if notes_hit_x("-x-x", quarter_notes):
        play(HAND_CLAP)
```

This is a pattern. Each pattern needs to be a function that takes no arguments and does not return anything. Instead the function is intended to follow this format:

```
If [particular point in time]:
    play([something])
```

Timing

This particular point in time has to be described as a boolean value. It has to be true or false. The program comes with several predefined variables that represent different points in time. Here is a list:

is_thirtytwo_note
is_sixteenth_note
is_eight_note
is_quarter_note
is_half_note
Is_beginning_of_bar (function - needs to be called)

These represent the beginnings of their respective note values. The following will sound like a punk rock bass line:

```
def bass():
    if is_eight_note:
        play(root)
```

You can also reference the current note value directly, by calling the following variables:

thirtytwo_notes
sixteenth_notes
eight_notes
quarter_notes
half_notes
relative_bar
bar

Here is the punk rock bass line using the above:

```
def bass():
    If eight_note in range(8):
        play(root)
```

This checks if the variable eight_note is a member of the list containing integers from 0 to eight and if it is, plays the root note of the current chord. The variables are decimal numbers so they will not stay a whole number and then tick up to the next one during the simulation of the song, except for thirtytwo notes which is the highest resolution time unit and also bar and relative_bar.

All notes except bar and relative_bar reset for each bar which lasts for 4 quarter notes.

relative_bar represents the bar within a sequence of four bars, so it resets at every four bars. This can be useful to reference if you want to trigger motifs that alternate between bars, but you don't want to use the modulo operator to figure out where you are.

bar represents the absolute number of bars within a part of a song. This will reset for a new part. A part here is defined as a list of chord progression and functions defined in the list of lists assigned to the PARTS variable at the end of the code.

There is yet another way of defining a particular point in time, through a function called notes_hit_x. It takes two arguments, a string of random characters and x-es which describe when you want to play something, and a a numerical value, typically one of the variables like eight_notes, quarters_notes and so on that is the value you want to check if is hitting x.

Here is the punk rock bass line as an example:

```
def bass():
    If notes_hit_x(“xxxxxxxx”, eight_notes):
        play(root)
```

Here is the same bass line with sixteenth_notes as the value you’re checking against:

```
def bass():
    If notes_hit_x(“x-x-x-x-x-x-x-x”,sixteenth_notes):
        play(root)
```

It achieves the same result.

The idea is to provide a visual and easy way to represent more complex rhythmical patterns without having to write a bunch of code.

Notes

After you’ve defined your point in time for something to play, you typically want to specify what to play.

You can represent notes as integers, or you can reference note objects that will eventually be evaluated to integers at the end of the song.

For drums, the integers are referenced directly since they don’t rely on the melodic information of the song. A list of variables that follow the midi drum “protocol” (e.g. add a drum vst to the track and it will play kick for C3 and so on) are available as all uppercase letters with space replaced with underscore such as the following:

```
def drums():
    if is_quarter_note:
        play(BASS_DRUM)
    if notes_hit_x("-x-x", quarter_notes):
        play(HAND_CLAP)
```

Here is a list of the available variables in natural language:

https://musescore.org/sites/musescore.org/files/General%20MIDI%20Standard%20Percussion%20Set%20Key%20Map.pdf

For melodic notes, you can reference integers directly, but if you want to utilise already computed notes for the chords and key in question, these are assigned to reserved variables that changes dynamically as the song progresses. They are as follows:

Chord notes:

root
third
fourth
fifth
sixth
seventh

Even though you are not specifying a chord with a quality that gives more than a triad, they are still available and will follow the key to determine what to give. If a chord overrides what the key typically gives you - let's say a C minor in a c major scale - a third will give you the third of the minor chord.

key notes:

DO
DI
RE
RI
MI
FA

FI
SOL
SI
LA
LI
TI

The key notes are defined in soflege and they start from the beginning of the key root. There are only major keys at this point, so that a song in A minor will resolve to C major since they shared the same set of notes. DO will therefore be C in both Am and C.

Both the chord notes and the soflege notes can be prepended with high_ or low_ (uppercase for soflege) if you want to go up or down and octave based on the pattern's frequency range.

Full chords

You can also access full chords (a list of notes) through the following functions:

get_current_chord
voice_lead_notes

get_current_chord returns a list of note objects in root position. voice_lead_notes returns the same notes in a potentially inverted position to allow for minimal movement of notes from one chord to another.

Note objects

The notes returned from the chord functions are objects that hold variables called step and octave, that represent the offset from the first note of the key (not chord) and an octave. It also has the functions .get_next and .get_previous that has the defaults arguments steps=1 and mode='diatonic'. These will return a note object higher and lower respectively for the given scale and the number of steps.

The play function

Notes being emitted by using the play function at a given point in time. The play function takes an integer, a note object or a list of the two. In addition it takes the arguments release and velocity which have 0.5 and 100 as default arguments. 0.5 is equal to an eight note and velocity goes from 0 to 127 and describes how hard the note is going to be hit.

Playing sequences

If you want to play sequences, this can be achieved by using the play_pattern function that also takes a list of notes, but will play it in a sequence. It takes a a list of notes as the first argument, a list of integers representing pauses in quarter notes as the second argument, and optionally a list of integers representing the release values for each note as the third argument.

To generate pauses you can use the get_pauses function which takes a string of characters where x represent when you want to play the various notes and a string representing the note value as the second argument (“eight_notes”, “quarter_notes” etc).

BPM

You can control the BPM through the reserved variable BPM. It’s not required and it will default to 120 if not present. You will also typically be able to override the BPM in the DAW.

Key

You can also control the key through the reserved variable KEY, but if not present it will be set automatically based on the chord progression.

Frequency ranges

To not have to deal with the octaves of notes, you can use decorators for the patterns that will make sure the notes falls within a given frequency range. The decorators are the following:

@lowest_range
@low_range

```
@high_range
@highest_range
```

If no decorator is supplied it will use a mid frequency range.

The decorators can be used like this:

```
@low_range
def bass():
    If notes_hit_x(“xxxxxxxx”, eight_notes):
        play(root)
```

These ranges only affect note objects assigned to the reserved variables and functions like root, third, get_current_chord() and so on.

Silence in tracks

If you want to remove an instrument for a given section, you can just use None as a value in the list containing the functions for the PARTS variable. Here is an example:

```
PARTS = [
        [chorus*2, [chord_stabs, None, bass]],
        [verse*2, [chord_stabs,  drums, bass]]
]
```

where the second track will have silence during the first part before being populated by drum notes in the second part.

Instruments

To add instruments to the tracks (piano defaults for all if not specified), assign a list of values to the reserved variable TRACK_INSTRUMENTS, by referencing the class Instrument and one of its class variables likes this:

```
TRACK_INSTRUMENTS = [
        Instrument.PIZZICATO_STRINGS,
        Instrument.DRUMS,
        Instrument.ELECTRIC_BASS_FINGER
]
```

The available class variables reflect the midi protocol like shown here and are the following:

DRUMS
ACOUSTIC_GRAND_PIANO
BRIGHT_ACOUSTIC_PIANO
ELECTRIC_GRAND_PIANO
HONKY_TONK_PIANO
ELECTRIC_PIANO_1
ELECTRIC_PIANO_2
HARPSICHORD
CLAVINET
CELESTA
GLOCKENSPIEL
MUSIC_BOX
VIBRAPHONE
MARIMBA
XYLOPHONE
TUBULAR_BELLS
DULCIMER
DRAWBAR_ORGAN
PERCUSSIVE_ORGAN
ROCK_ORGAN
CHURCH_ORGAN
REED_ORGAN
ACCORDION
HARMONICA
TANGO_ACCORDION
ACOUSTIC_GUITAR_NYLON
ACOUSTIC_GUITAR_STEEL
ELECTRIC_GUITAR_JAZZ
ELECTRIC_GUITAR_CLEAN
ELECTRIC_GUITAR_MUTED
OVERDRIVEN_GUITAR
DISTORTION_GUITAR
GUITAR_HARMONICS
ACOUSTIC_BASS
ELECTRIC_BASS_FINGER
ELECTRIC_BASS_PICK
FRETLESS_BASS
SLAP_BASS_1
SLAP_BASS_2
SYNTH_BASS_1
SYNTH_BASS_2
VIOLIN

VIOLA
CELLO
CONTRABASS
TREMOLO_STRINGS
PIZZICATO_STRINGS
ORCHESTRAL_HARP
TIMPANI
STRING_ENSEMBLE_1
STRING_ENSEMBLE_2
SYNTH_STRINGS_1
SYNTH_STRINGS_2
CHOIR_AAHS
VOICE_OOHS
SYNTH_VOICE
ORCHESTRA_HIT
TRUMPET
TROMBONE
TUBA
MUTED_TRUMPET
FRENCH_HORN
BRASS_SECTION
SYNTH_BRASS_1
SYNTH_BRASS_2
SOPRANO_SAX
ALTO_SAX
TENOR_SAX
BARITONE_SAX
OBOE
ENGLISH_HORN
BASSOON
CLARINET
PICCOLO
FLUTE
RECORDER
PAN_FLUTE
BLOWN_BOTTLE
SHAKUHACHI
WHISTLE
OCARINA
LEAD_1_SQUARE
LEAD_2_SAWTOOTH
LEAD_3_CALLIOPE
LEAD_4_CHIFF
LEAD_5_CHARANG

LEAD_6_VOICE
LEAD_7_FIFTHS
LEAD_8_BASS_LEAD
PAD_1_NEW_AGE
PAD_2_WARM
PAD_3_POLYSYNTH
PAD_4_CHOIR
PAD_5_BOWED
PAD_6_METALLIC
PAD_7_HALO
PAD_8_SWEEP
FX_1_RAIN
FX_2_SOUNDTRACK
FX_3_CRYSTAL
FX_4_ATMOSPHERE
FX_5_BRIGHTNESS
FX_6_GOBLINS
FX_7_ECHOES
FX_8_SCI_FI
SITAR
BANJO
SHAMISEN
KOTO
KALIMBA
BAG_PIPE
FIDDLE
SHANAI
TINKLE_BELL
AGOGO
STEEL_DRUMS
WOODBLOCK
TAIKO_DRUM
MELODIC_TOM
SYNTH_DRUM
REVERSE_CYMBAL
GUITAR_FRET_NOISE
BREATH_NOISE
SEASHORE
BIRD_TWEET
TELEPHONE_RING
HELICOPTER
APPLAUSE
GUNSHOT