Assignment 2

Overview

In lecture, we learned about audio data stored in WAVE files, how to read them, how to play them in full, and how to play them as snippets.

Your assignment is to flesh out the basic interfaces that we developed in class with more features. Then, put these to good use by creating a mini performance system.
Note that the assignment code contains *superstition.mp3* but not *superstition.wav* (so the download would not be too long). If you want to play with the same code as we saw in Lecture, you should first convert the mp3 to a wave file using Sonic Visualizer (or whatever conversion tool you want).

Part 0

Find some audio - a song that you want to play with - and convert it to a WAV file with Sonic Visualizer.

Part 1

Add shuttle controls to *WaveFileGenerator*. Allow the user to:

- Toggle between playing and pausing the generator. On your music player, this is usually connected to the "Play/Pause Button"
- Allow the user to reset back to the beginning of the file. On your music player, this is usually connected to the "Reset Button."

In other words, fill this in:

```
class WaveFileGenerator(object):
   ...

   def play_toggle(self):
      ...

   def reset(self):
      ...
```

Part 2

Create some test regions in the WAV file using Sonic Visualizer:

- Open the WAVE file
- Add a new Regions Layer
- Create a few regions that you want to convert to Snippets.
- Export the regions layer and call it <name>_regions.txt

Write the code to read this text file so that you can easily create Wave Snippets from the data file.
Make a class *AudioRegion* that holds the data for one Region. It should have a name, a start frame, and a length.

Make a class *SongRegions* that holds a collection of *AudioRegions* and knows how to create that collection by parsing the _regions.txt file.

Finally, create a python dictionary of *WaveSnippets* from the *SongRegions*. The dictionary should be of the format:

```
{
   name1: snippet1,
   name2: snippet2
}
```

where *nameN* is a string and *snippetN* is a *WaveSnippet* instance. Now, you can easily play back any of your snippets by retrieving a snippet by name, creating a generator, and handing it off to the Audio class.

Part 3

Add a looping option to WaveSnippet's generator. If looping is enabled, when the generator gets to the end of the audio data, it will automatically loop back to the beginning. So the make_generator factory function should look like this:

```
def make_generator(self, loop):
   ...
```

But this means that if looping is on, the generator will never stop producing audio, so add a function to stop the generator immediately:

```
class Generator(object):
   ...
   def stop(self):
      ...
```

In fact, this stop function should work even if looping is not turned on.

You can test stopping by hooking into keyboard keys. I suggest trying to use "key release" to stop a WaveSnippet generator that was started with a "key press". In kivy, create and fill out this function:

```
def on_key_up(self, keycode):
   ...
```

When you author regions for looping in Sonic Visualizer, you'll need to pay careful attention to the start and end of the region. Use the built-in playback looping ability to test out what the region sounds like when it loops.

Part 4

So far, all of our audio has played back at the intended speed (ie, the speed that it was recorded). Let's mess with that as well! For WaveSnippet's generator, add the ability to play back the audio at a different speed. The generator should take an additional argument, *speed*. Plus, you should be able to change the speed on the fly:

```
def make_generator(self, loop, speed = 1.0):
   ...

class Generator(object):
   ...

   def set_speed(self, speed):
   ...
```

Implementing speed change requires resampling the audio using interpolation. We discussed how interpolation works in class, but implementing it is a job for you. Remember the function *np.interp*. It will come in handy. You might also find np.*linspace* useful.

Part 5

Now, pull it all together to make a mini-performance system that combines some or all of the systems you have built so far:

- Full song playback
- Snippet playback
- Looping
- Speed changes
- Note synthesis

Find song / audio content you like, create mappings to trigger that content in different ways, and if you like, add mappings for synth note playback on top. You can build on the work you did in Assignment 1, or start with something else.

Some ideas:

- Drum loops are fun. Find audio of drums and loop those
- Short audio snippets that only contain one note or chord can be triggered as notes. If you carefully alter their speed, you can get a variety of different pitches as well.
- Loop a longer snippet and trigger short snippets on top of that. This works well if the music of the long snippet acts as background and the short snippets can act as melody or foreground music.
- Identify the chords / harmonic progression of some areas of a song and make note mappings that match the harmonies and can be played on top of those areas.

Briefly document how to control your system in a README file.
Create a short video recording of you performing your creation. This need not be long. 30-60 seconds is fine. You can upload the video to YouTube (or any video sharing site) and send me a link (preferred), or include the video in the submission. Remember to also include the audio file itself: wave file or mp3 file (which I can convert to wave if needed).

Finally...

Remember to add a paragraph about how you collaborated. Please have good comments in your code. When submitting your solution, submit a zip file that has all the necessary files. For example, if you used other files that I provided (like core.py), re-provide those files back to me in your submission. Upload your zip file to Stellar / Homework.