COMP151 - Project 2 - Get your Synth On.

Fall 2020

Your final project is all about sound synthesis, song writing, and beat creation. By the time it's done you'll invent a new synthesized sound, program some beats, and program a song. Like your first project, you'll be directed to complete a series of pre-designated versions, each of which will increase your potential grade.

You'll be given a small module called *waveGen.py* and a short program that shows you how to use it. The waveGen module provides functions for generating white noise, sine waves, square waves, and triangle waves. With it you can generate base waves that you can chop, mix, modify, and otherwise turn into something new and original. As as bonus, you get to explore how programmers manage larger programs by using code spread across multiple files with things like modules.

Version 0

The first version of your program is really about setting up a simple musical sequence with basic waves that you can then begin to enhance with newly invented synth sounds. You have two main goals, collect useful sound manipulation functions that you've already encountered and setup a basic main program.

- Get the starter program working by modifying the *addLibPath* line to point towards the directory in which you've saved waveGen.py.
- Add your solutions to homework problem 8.7 (*clip*, *copy*, and *reverse*) to your project. If you don't have working solutions to these problems, then you may use the non-array notation based versions in chapter 8. Feel free to grab other effects (shift and echo come to mind) if you think you'll want to use them. You're allowed to copy and past the code into your project file, but I encourage you to instead add your homework code to the same directory as waveGen.py and import it as a module. If you do this, then your homework file should contain only definitions and no test code that executes when you hit load.
- Have a well defined main function that coordinates the following functionality:
  - With the help of the copy function, create 8 beats of music at 200 bpm using the following 8 note sequence: A3, C4, E4, G4, A4, G4, E4, C4. Each note should last for half a beat and occur at the beginning of each beat. You can choose whichever wave form and amplitude you'd like. Feel free to mix and match if you want. This note sequence is really just meant to give you a good test case for your future synth.
  - Open the result to explore/play the music.

A complete version 0 program will, when run, pop up the explore window showing you the wave. There should be a clear visual difference between each note due to the difference in frequencies. In some of the resources below you'll find a link and some discussion about note names and the frequency of specific notes.

Version 0.5

In this version you'll create a *synth* and modify the result of version 0 to play the same note pattern using that synth rather than a pure wave. For our purposes, a synth is just a basic wave form that is modified in some way in order to produce a new and different waveform. For melodic, musical voices we will want to design a function that works in the same way as a wave generator: you provide frequency, amplitude, and duration (in seconds) and it returns a wave with more or less those characteristics. For example, *mysynth(440.0,10000,2)* would produce something in the ballpark of an A4 (440 Hz) at amplitude 10000 that lasts for 2 seconds.

Your first synth should take an additive synthesis approach and mix small amounts of other tones in with the fundamental frequency. The standard thing to use sine waves and add *overtones*, i.e. integer multiples of the fundamental. For example, *mysynth(440.0,10000,2)* might take the fundamental frequency of 440.0 Hz and add in small amounts, i.e. lower amplitudes, of 880.0 Hz and 1320.0 Hz because they are double and triple the fundamental. You can try mixing other frequencies and other waveforms as well. Have fun with it. The mixer effect lets you start with waves of similar amplitude and boost or lower their amplitudes as you combine them.

- Design a function called *mixer* that works as described in the effects section below.

- Use *mixer* as the basis for a function called *mysynth* that produces a mixture of at least two waves. Try things like combining different wave-forms of the same frequency or mixing in small amounts of a different frequencies to the main frequency. You can even try mixing noise in with a note to see what happens. Have fun with it. Play around. Find some dope sounds.
- Modify your program so that it plays the same music but does so with the output of *mysynth* rather than a basic waveform.

Version 0.75

New version. New sound effect. Updated Synth.

- Write a function called *envelope* that implements the envelope effect described in the effects section below.
- Add an envolope to *mysynth*.
- Modify your program so that in addition to the original melody there is an A2 on every other beat for 8 beats. That's the first, third, fifth, and seventh beat. Because the new part is very repetitive, *use a loop to create the 4 note pattern*. In order to get the best result, you'll want to first construct each part on its own and then mix them together. Imagine you're recreating a band. Each part is an instrument, can be played on its own, and is mixed together to create a song.

Version 1.0

Time for some beats.

- Design a new synth called *drum*. It should be based on white noise (think snare drum) or some very low frequencies (think kick drum) and have a very short envelope. Because these synths are percussive, their pitch doesn't change, the drum function should take only two inputs: amplitude and duration.
- Construct a drum beat to go with your song by coming up with a two beat or four beat pattern that repeats for a total of 8 beats. Here's an easy way to make a 2 beat pattern. Divide each beat into half so that two beats gives you 4 half beats. For example, if a beat takes 0.5 seconds, then the half beats are each 0.25 seconds. Now, just choose some, but not all, of those half beats and sound your *drum* at those times leaving silence during the unused half beats. Repeat that pattern again three more times and boom, 8 beat rhythm. You can also get some good beats by breaking up a beat into quarters giving you 8 quarter beats within a 2 beat time frame.
- Modify your program to add your drum part to the song. Be sure to use the same techinque as before. Constuct each part individually and then mix them together.
- Make it so your program saves this sound to a .wav file on the hard drive.

Version 2.0

Play time! Use your synth and your drum, or invent new sounds, and construct a 32 beat song. The only requirements are that it involves at least 3 parts, like our little tune from version 1.0, and that it's 32 beats long. It can be purely percussive if you want. It does not need to involve notes. The key to making this musical, rhythmic, and not too tedious is repetition. Repetition allows programmable patterns and is also one of the cornerstones of music. If you need some more structure, then I recommend this approach:

1. Come up 2, 4 beat sequences, each with 3 parts. We'll call the first part the **A** part and the second the **B** part.
2. Your song is just the **A** part twice, then the **B** part, then **A** again.
   This is called an AABA song. It's a very well documented song structure and you can find lots of examples and variations by searching the web for *AABA song form*.

Grades and Dates

We'll dedicate two lab periods and some class time to this project. Use that time wisely. The final grade for the project is determined by how many versions you complete and the quality of the code you submit.

Important Dates

Date	Event
11/9-10	Lab time to begin project

Date	Event
11/13	**Version 0 Due**
11/16-17	Lab time for the project
11/23-24	Lab time to continue project
12/24	**Project Due**

Project code should be submitted to GradeScope by 11:59 midnight on 12/24. Your sound files (version 1 and 2) should be uploaded to Moodle.

Grades

Your grade is determined based on the highest version number you complete (see the chart below) and the overall quality of the work submitted.

Version	Grade
0	D
0.5	C
0.75	C+
1.0	B
2.0	A

Quality is measured by the cleanliness and clarity of the code and the appropriate use of sub-functions that exhibit reusable design. The versions determine the range in which you grade will fall and the quality of your code will determine where your grade lands within that range. For example, let's say you've finished version 0.75 and your code is neatly presented and well designed. Your grade starts at a C+ due to the version that you're presenting and could go up as high as a low B- based on the quality. Conversely, if your code is all crammed into a single function and your variables are poorly named, then your grade could go down as low as a high C due to poor quality.

Resources on Music and Audio Synthesis

From Notes to Frequencies

Western music, and the music you'll be programming, is based on a clear 12 note pattern that repeats itself over and over at higher and higher pitches. This repetition produces a standard set of notes from which all music is written. You can find their names and the wave frequencies that go with them here: Note Names to Wave Frequency. At the top of the chart is C0, a very low note. The 12 note pattern starting at C0 continues down the chart to B0 and then begins the pattern again at C1. We tend to like notes around C4 and place this note in the middle of a piano or keyboard. For this reason it's called *middle C*.

Scales, Intervals, and Resonant Tones

The difference between C0 and C1, or any note and the next higher numbered form of that note, is called an *octave*. The numbers you see on note names are their octave numbers. From a frequency perspective, moving up an octave means to double the frequency. For example, notice A0 is 27.5, A1 is 55.0, A2 is 110.0, A3 is 220.0, and so on. There is also a set of fixed multiples between a note and the other 11 notes within the octave. Here you see an octave starting on C.

Note	Frequency (Ratio)	Frequency (Decimal)
C	f	f

Note	Frequency (Ratio)	Frequency (Decimal)
C#	(256/243)f	1.053f
D	(9/8)f	1.125f
D#	(32/27)f	1.185f
E	(81/64)f	1.266f
F	(4/3)f	1.333f
F#	(729/512)f	1.424f
G	(3/2)f	1.5f
G#	(128/81)f	1.58f
A	(27/16)f	1.687f
A#	(16/9)f	1.778f
B	(243/128)f	1.898f
C	2f	2f

While the chart above starts on C, it actually doesn't matter where you start. If you start at a G and call its frequency f, then the 12 note sequence up to the next G follows this same pattern of multiples as moving from C to C.

Where this becomes useful is in mixing in related notes to a target dominant tone of a voice. Let's say your voice is meant to produce some note with frequency *f* (maybe 440). You might mix in just a little bit the fourth (5 steps up) or fifth above that note (6 steps up). You just take your dominant frequency *f* and multiply by (4/3) and/or (3/2) to get the right note for you mix. These notes tend to be very resonant with your dominant frequency and work well in combination with that note. On our C-based chart this is the equivalent of mixing F and G with C.

Sound Effects

Below you'll find descriptions of each of the key effects you've been asked to implement for this project. Be sure you follow the specifications given below.

Mixer

The mixer takes two audio signals and combines them in user-specified amounts that determine the relative proportion of each sound in the mix. This results in a function that takes two sound objects and two numbers between 0 and 1 and returns a mixture of those sounds based on the values of the numbers. If you use the number 1, then you're asking for 100% of the original sound. Using 0 then means using none of the original sound. In this way we can not only control the proportion of each sound in the result but the overall volume as well. Finally, this mixer should have no problems mixing sounds of differing lengths. The result should be as long as the longest sound. If your first sound is two seconds and the second is one second, then the result is two seconds where the first second is a mix of the two sounds and the last second is just the last second of the first sound.

Let's explore some examples. Suppose you have two sounds: *s1* and *s2*. Doing something like *mixer(s1,1,s2,1)* would mix them together in equal parts and at 100% of their original volume. On the other hand doing *mixer(s1,.5,s2,.5)* cuts the volume by half on each sound but then mixes them together. By keeping the numbers the same we can keep the mix balanced but cut back the volume to avoid clipping that is likely to occur if the original amplitude of *s1* and *s1* was already pretty high.

If instead we did *mixer(s1,1,s2,.5)* then we'd mix the full volume of s1 with s2 at half volume. This lets us keep s1 as a dominant wave and use s2 to spice it up, or color it, a bit. This kind of mixing is also important if we want to mix more than two sounds in equal proportions. Let's say we also have sound *s3* and would like to mix it equally with *s1* and *s2*. We can first do *temp = *mixer(s1,0.5,s2,0.5)* to get an equal mix of *s1* and *s2*. If we then did *mixer(temp,0.5,s3,0.5)* we'd end up with too much of the temp sound because we've effectively cut *s1* and *s2* in half again. What we need to do is *mixer(temp,0.667,s3,0.333)* to account for the fact that two thirds of the components are mixed in *temp* and only one third in *s3*. A similar kind of scaling would need to occur to mix in a fourth signal in equal proportion to three equally mixed signals. For this project you're welcome to find sounds and mixes that you like through trial and error and now worry about the overall proportions. If, however, you find that you have too much or too little of a sound in a mix containing more than two sounds, then keep this discussion in mind.

Attack-Decay Envelope

Natural sounds do not typically start at one amplitude and stay at that amplitude for as long as you'd like them to. This is, however, the kind of sound our wave generators produce. A naturally occurring sound typically has a volume *envelope* where the volume grows to a peak and then shrinks back to nothing. The growth phase is called the *attack*. The shrinking phase is called the *decay*. A basic envelope effects lets you set the length of attack and decay.

Your *envelope* function should take a sound object, an attack length in seconds, and a decay length in seconds. It should then increase the amplitude of the sound from 0% to 100% for the duration of the attack phase and then decrease from 100% back to 0% for the duration of the decay phase. The catch here is that the input sound does not need to be as long as the sum of the attack and decay time. Imagine banging a gong. It has a short attack and a long decay. If you hit it once it and leave it alone you'll hear a loud tone that slowly decays to nothing. If you hit it again before it stops ringing, then you are effectively stopping the old envelope mid decay and starting a new envelope. Your *envelope* function should support this kind of musical sound by computing as much of the envelope as fits the given sound. Suppose your envelope has an attack of 0.25 seconds and a decay of 1 second. If the given sound is 0.5 seconds, then you get all of the attack phase and only 0.25 seconds of decay. If the given sound is 2 seconds you get the full envelope including 0.75 seconds of silence created at the end of the sound.