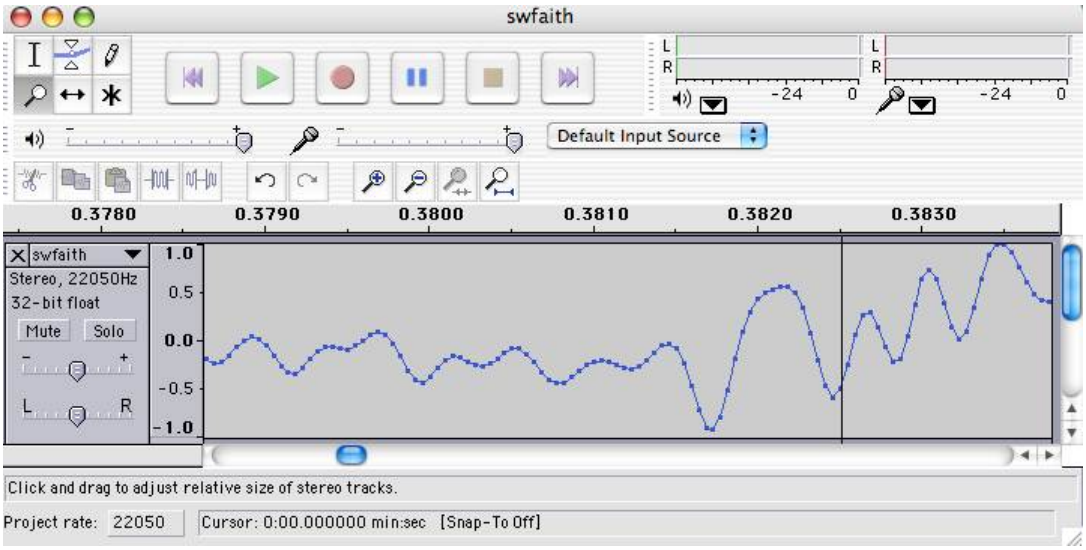
Before proceeding, please read the accompanying sound_waves.pdf document.

## Background on representing audio information

What is inside an audio file? Depending on the format, the actual audio data might be encoded in many different ways. One of the most basic is known as pulse code modulation (PCM), in which the sound waves are sampled repeatedly and given values in the range -128 to 127 (if 1 byte per sound sample is used) or -32768 to 32767 (if there are 2 bytes for each sample). Wikipedia explains it here. The **.wav** file format essentially encodes audio in this way, and the cross-platform program Audacity is an excellent tool for visualizing the individual PCM samples of an audio file. You don't need Audacity for this problem, but it runs on Windows and Macs and is fun to play around with if you'd like to. Audacity can also convert to `.wav` from `.mp3` and many other formats.




## Getting started

- Download python-sound.zip and unpack it.
- Test/run the provided `wavmod.py` file. Make sure it runs without error. (No output yet.)
- Read through the comments and methods of the entire module. Note that there are comments that even describe functions that are not part of wavmod, but are imported from a module called csaudio.
- Create a main function in which you create a WavMod object using the 'swfaith.wav' file that has been provided to you, and call the `test()` method on it. When you run your program the sound file should play. (Make sure your volume is turned up so you can hear it.)
- If that works, you should be able to test the two already-written methods (`changeSpeed` and `flipflop`).
    - Try out `changeSpeed` with the provided files and several different speeds, e.g.,
      ```
      changeSpeed(30000)
      changeSpeed(14000)
      ```
    - Try out `flipflop`, as well.
- **Warning**: if you `return` the list of raw sound data (notice these `return` statements are currently commented out…), and you call these functions from the IDLE interactive Shell, be **sure** to assign the result to a variable! By doing so, e.g., `aList = wm.changeSpeed(30000)`, IDLE will not try to display all of the raw sound data. If you don't assign the call to a variable, IDLE *will* try to print out the raw sound data, it will be too big, and IDLE will hang.

  Thus, when using the `return` statements, the assignment to `aList` is important, because these functions return a ***large*** list of sound data samples! You can always look at a portion of the sound data in `aList` using list slicing, e.g., `aList[0:100]`.
- Read over each function *and its comments and docstrings* to begin getting a sense of what is happening to the sound data in each case. From the commented explanation of the code for the changeSpeed method, you should be able to understand the code of both methods. Let us know if you have any questions!!

## Methods you need to write

### *Question 1: Sound reversal*

- Write a method `reverse()` that will write out a new .wav file in which the order of the sound samples has been reversed. As in `changeSpeed` and `flipflop`, your `reverse` function should play the sound after writing it to file and it should return the new sound's list of raw data samples. Use `flipflop` as a model. In particular, don't implement this function (or any sound-processing

function) using raw recursion—sounds are so large that Python will run out of memory even for very short sounds. You may comment out the return statement as we have done in `flipflop`.

### *Question 2: Whisper/shout mode*

- Write a method `volume(fraction)` that takes in a `.wav` file's name (as a string) and a floating-point value named `fraction`, which represents how much louder or softer the output sound should be. For example,
    - `volume(0.5)` should create a sound with half the amplitude of the original
    - `volume(1.5)` should create a sound with one and a half the amplitude of the original.

  As with `flipflop` and `reverse`, your `volume` function should play the sound after writing it to file and it should return the new sound's list of raw data samples. The output sound's fileName, `newFile` is handled in the usual way.

  You'll notice that your hearing adjusts remarkably well to this function's changes in absolute volume, making their perceived effect less than you might expect. You will also find that if you increase the volume too much, the sound becomes distorted, just as when you turn an amplifier up to 11. =)

### *Question 3: A pure-tone generator*

- Write a method `oneFreq(freq)` that takes in a frequency (in hertz, or periods per second) and it writes out a .wav file (you choose the name) that is one second of a *cosine* wave — a pure tone — at the input frequency. I used `math.cos`. As usual, the `oneFreq` function should play the sound generated and return the list of its raw sample data. The sample rate used to create the new sound's file should be 22050 samples per second. The amplitude of your cosine wave should be the maximum of 32767.0.

  At the heart of this function will probably be a loop that looks like this: `for i in range(NUM_SAMPLES)`. Variables that I defined for use in this expression included
    - `amplitude`, the amplitude of the cosine wave
    - `twoPi`, the value of 2 * pi
    - `sampleRate`, the sampling rate used

### *Question 4: Chords*

- Certainly `oneFreq` is not enough! Write a method `multiFreq(freqList)` whose input named `freqList` is a three-element list of frequencies, e.g.,

  ```
  [freq0, freq1, freq2]
  ```

  The `multiFreq` function should write out a two-second chord with all of those frequencies. Again, use 22050 hz as the sample rate. You can use your method `oneFreq` to do the "heavy lifting" for you here — just modify it so that it returns the list of samples to you for the one tone, and then call it three times, once for each tone/frequency. Also, add a parameter for how many seconds the returned sound should last. Interestingly enough, to generate the sound of a chord that plays the three notes at once, for each sample, you simply *average* the corresponding sample from the three individual sounds!

  For those who have not studied the mathematics of music, it is useful to know that a "half step" on the piano is equal to a frequency ratio of the twelfth root of two, or (as Python happily informs us) 1.05946309436. A whole step's ratio is the product of two half steps, or 1.122462048309373, a minor third's ratio is a half step to the third power, or 1.1892071150027212, and a major third's ratio is 1.2599210498948734. If you don't have musical training, it's also worth knowing that concert "A" is 440 Hz, and a pleasing chord can be made from a root (1.0), a major third, and a perfect fifth (which ideally is a ratio of 1.5, although you can see by multiplying half steps that you can't quite hit that in our usual tuning system).

  So `multiFreq([440, 440 * 1.26, 440 * 1.5])` should sound fairly nice to your ears.

  You may wish to experiment with "odd" frequencies to see what kind of sounds you get. Try `multiFreq([440, 442, 443])` for example.

  Unfortunately, it usually takes at least 6 to 8 frequencies in ratios of 1.5-to-1 or 2-to-1 to begin to simulate the sound of acoustic instruments, and we won't go quite that far with this function.

  You can make this method more sophisticated by making the input parameter `freqList` hold not only three frequencies, but their respective "power" levels, so that each note can have a different volume.

### *Question 5: Your own effect(s)…*

- Write one or more methods that combine or extend these effects—or, simply invent your own audio-altering computation. Be sure to include a comment with your function and/or in its docstring describing what your effect is. We look forward to trying your function on the sounds we have!