Week 3 Lab: *Sounds good!*

Copied from: https://www.cs.hmc.edu/twiki/bin/view/CS5/SoundsGoodLab on 3/22/2017

[35 pts; individual or pair; filename: `hw3pr1.py`]

Starting files to download

Starter file for this problem:

Download pythonSounds.zip from this link.

Be sure to unzip that folder somewhere. It has several files ***all of which need to stay in that same folder***:

- `hw3pr1.py` (the file to edit and run!)
- `swfaith.wav`
- `swnotry.wav`
- `spam.wav`
- `csaudio.py`
- `play` (A small MacOS application for playing sounds...)

You need to work from within this folder!

- To do this, be sure to `cd` into the `pythonSounds` folder.
- For example, you could first move the *whole* `pythonSounds` folder to your desktop...
- Then `cd Desktop` and `cd pythonSounds`
- From there, you can run the usual `ipython` and then `run hw3pr1.py`
- Please *do* keep the files inside `pythonSounds` all together!

If you move the `hw3pr1.py` file without the others..., things won't work!

Warm-up: helper functions using list comprehensions (*LC*s)

Take a moment to remind yourself how *list comprehensions* work... .

Look over the `three_ize` function near the top of the `hw3pr1.py` file:

```
def three_ize( L ):
    """ three_ize is the motto of the green CS 5 alien
        it's also a function that takes in a list and
        returns a list of elements each three times as large
    """
    # this is an example of a list comprehension
    LC = [ 3 * x for x in L ]
    return LC
```

This function "maps" the expression `3*x` over the values `x` in the list `L`.

Try it out with

Example(s):

```
In [1]: three_ize( [13, 14, 15] )
Out[1]: [39, 42, 45]
```

List comprehensions are a versatile syntax for mapping a function (or expression) across all elements of a list.

If you feel good about list comprehensions, onward! (If you think more explanation/practice would be worthwhile, try our ListComprehension page.)

Function to write #1: `scale`

With the above function as your model, write a function `scale` with the following signature:

```
def scale(L, scale_factor):
```

where `scale` returns a list similar to `L`, except that each element has been multiplied by `scale_factor`.

Example(s):

```
In [1]: scale( [70, 80, 420], 0.1 )
Out[1]: [7.0, 8.0, 42.0]
```

Use a list comprehension here.

Going further: *index-based* list comprehensions

Next, make sure this `three_ize_by_index` function is in your `hw3pr1.py` file.

Look it over:

```
def three_ize_by_index( L ):
    """ three_ize_by_i has the same I/O behavior as three_ize
        but it uses the INDEX of each element, instead of
        using the elements themselves -- this is much more
flexible!
    """
    # another example of a list comprehension
    N = len(L)
    LC = [ 3 * L[i] for i in range(N) ]
    return LC
```

This function does *exactly the same thing* as `three_ize`—it simply uses the index of each element to do so. That is, now the ***location*** of each element, named `i`, is changing

This index-based use of list comprehensions is even more flexible than the element-based style, as the next couple of questions will show.

Functions to write #2 and #3: `add_2` and `add_3`

With the above ***index-based*** functions as a guide, write a function `add_2` with the following signature:

```
def add_2(L, M):
```

such that `add_2` takes in two lists and returns a single list that is an element-by-element sum of the two arguments. If the arguments are different lengths, your `add_2` should return a list that is as long as

the *shorter* of the two. Just ignore or drop the extra elements from the longer list.

Using `min` and `len(L)` and `len(M)` together is one way to do this. For example, the line

```
N = min( len(L), len(M) )
```

will assign `N` to the smaller of the lengths of `L` and `M`.

You will want to use the *index-based* approach for this `add_2` function. You might use `three_ize_by_index` as a starting point…. Also, consider how this `LC` might help:

```
LC = [  L[i]+M[i] for      ...              ]
```

Here are two examples of `add_2` in action:

```
In [1]: add_2( [10, 11, 12], [20, 25, 30] )
Out[1]: [30, 36, 42]

In [2]: add_2( [10, 11], [20, 25, 30] )
Out[2]: [30, 36]
```

Then, write the analogous three-input function `add_3` with the following signature:

```
def add_3(L, M, P):
```

where `L`, `M`, and `P` are all lists and `add_3` outputs the sum of all of them, but only as many elements as the shortest among them has.

The strategy will be very similar to `add_2`.

Function to write #4: `add_scale_2`

Next, write a function `add_scale_2` with the following signature:

```
def add_scale_2(L, M, L_scale, M_scale):
```

such that `add_scale_2` takes in two lists `L` and `M` and two floating-point numbers `L_scale` and `M_scale`. These stand for *scale for L* and *scale for M*, respectively.

Then, `add_scale_2` should return a single list that is an element-by-element sum of the two inputs, *each scaled by its respective floating-point value*. If the inputs are different lengths, your `add_scale_2` should return a list that is as long as the *shorter* of the two. Again, just drop any extra elements.

Example(s):

```
In [1]: add_scale_2( [10, 20, 30], [7, 8, 9], 0.1, 10 )
Out[1]: [71.0, 82.0, 93.0]

In [2]: add_scale_2( [10, 20, 30], [7, 8], 0.1, 10 )
Out[2]: [71.0, 82.0]
```

This will not be too different from the previous examples!

A helper function: `randomize`

Next, take a look at this function in your `hw3pr1.py` file:

```
def randomize( x, chance_of_replacing ):
    """ randomize takes in an original value, x
        and a fraction named chance_of_replacing

        With the "chance_of_replacing" chance, it
        should return a random float from -32767 to 32767

        Otherwise, it should return x (not replacing it)
    """
    r = random.uniform(0,1)
    if r < chance_of_replacing:
        return random.uniform(-32768,32767)
    else:
        return x
```

Read over the docstring and try it out.

Nothing to do here except build an understanding of what this function is doing: how often it returns the original input and how often it returns a

random value. That random value happens to always be within the amplitude of a sound's pressure samples.

Though it's random, here is a set of five real runs:

```
In [1]: randomize(42, .5)
Out[1]: 42

In [2]: randomize(42, .5)
Out[2]: 42

In [3]: randomize(42, .5)
Out[3]: 29209.30669767395

In [4]: randomize(42, .5)
Out[4]: 42

In [5]: randomize(42, .5)
Out[5]: 17751.221299744262
```

Function to write #5: `replace_some`

Next, write a function `replace_some` with the following signature:

```
def replace_some(L, chance_of_replacing):
```

such that `replace_some` takes in a list `L` and a floating-point value `chance_of_replacing`.

Then, `replace_some` should independently replace—or not replace—each element in `L`, using the helper function `randomize`.

Since this function is random, the runs below won't be replicated on your system, but try yours out to make sure it's working in a similar fashion.

Hint: use `randomize` in a list comprehension: that's it! Consider how to complete this thought (and don't forget to return `LC`):

```
    LC = [ randomize( ____, _________ ) for x in L ]
```

Example(s):

```
In [1]: replace_some( range(40,50), .5 )    # replace about half
(hopefully the 42 remains!)
```

```
Out[1]: [40, 41, 42, -17461.09350529409, 44, -
13989.513742241645, 46, -26247.774200304026, 48, 49]

In [2]: replace_some( range(20,30), .1 )   # replace about a
tenth  (but it's random: here 2 of them get replaced)
Out[2]: [20, 21, 16774.26240973895, 23, 24, 25, -
18184.919872079583, 27, 28, 29]
```

In addition to providing practice with data and functions, the above examples will be helpful in creating functions that handle audio data in various ways... .

The `replace_some` function will allow you to add "static" (random values) to *some* of any sound, e.g., to make it sound "crackly."

***Sound* coding...**

First things first: try out this function, which should already be in your `hw3pr1.py` file.

You can run it with `test()`:

```
# a function to make sure everything is working
def test():
    """ a test function that plays swfaith.wav
        You'll need swfailt.wav in this folder.
    """
    play( 'swfaith.wav' )
```

For this to work, your Python will need to support sound (every version we've tested does). If yours does not—no problem, simply work with a partner from here on during this lab.

Also, you'll need the `swfaith.wav` file in the folder in which `hw3pr1.py` is located. As long as you're in the original folder, all of this should be the case. If not, go grab all of those files that came with `hw3pr1.py` and copy them over to whichever folder you're working in.

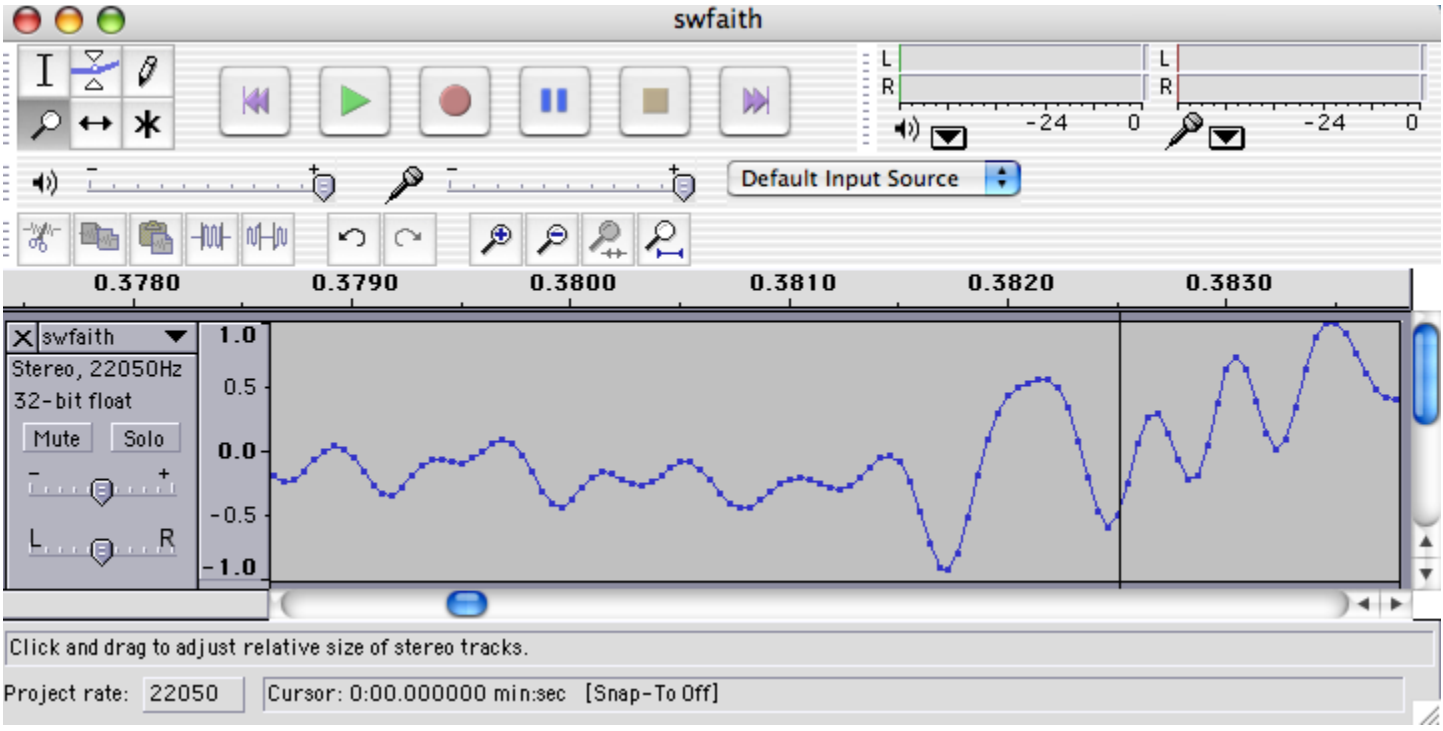
Before we go on, you'll need a bit of background information on audio data. Then you'll have a chance to write a number of audio-processing functions.

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 every so often 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 encodes audio in basically 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. Last but not least, Audacity was created by Dominic Mazzoni, an HMC alum!



Getting started with sound

We present two examples to start acquiring and manipulating sound data. Try these:

Sound example #1: changeSpeed

This function should already be in your file, but if not, it's here for easy copy-and-paste:

```
# The example changeSpeed function
def changeSpeed(filename, newsr):
    """ changeSpeed allows the user to change an audio file's
speed
        input: filename, the name of the original file
               newsr, the new sampling rate in samples per
second
        output: no return value, but
                this creates the sound file 'out.wav'
                and plays it
    """
    print("Playing the original sound...")
    play(filename)

    sound_data = [0,0]             # an "empty" list
    read_wav(filename,sound_data) # get data INTO sound_data

    samps = sound_data[0]          # the raw pressure samples

    print( "The first 10 sound-pressure samples are\n",
samps[:10])
    sr = sound_data[1]             # the sampling rate, sr

    print( "The number of samples per second is", sr)

    # we don't really need this line, but for consistency...
    newsamps = samps                        # same samples as
before
    new_sound_data = [ newsamps, newsr ]   # new sound data pair
    write_wav( new_sound_data, "out.wav" ) # write data to
out.wav
    print("\nPlaying new sound...")
    play( 'out.wav' )   # play the new file, 'out.wav'
```

Read over this example and try it out on the three sound files provided:

```
In [1]: changeSpeed( "swfaith.wav", 44100 ) # fast Vader
... some printing ...
```

```
In [2]: changeSpeed( "spam.wav", 11025 )  # slow Monty Python
... some printing ...

In [3]: changeSpeed( "swnotry.wav", 22050 ) # regular-speed Yoda
... some printing ...
```

Those lines should already be there. If you're using ipython notebook, please place them in a cell and then run that cell.

Before continuing, it's good to review some points about the code and how it works:

1. The sound data is returned by the call to `read_wav` in two parts using the call

   ```
   samps, sr = read_wav(filename)
   ```

   Python is nice in that you can return any number of values from a function. Here, `readwav` is returning two values.

2. After that call, the variable `samps` should have a large list of raw pressure samples (floats). **Don't print this list**—it can be too big and can slow down or choke the ipython shell!
3. Also, after that call, the variable `sr` is an integer that represents the *sampling rate*, i.e., the number of samples that should be played per second for normal-speed playback.
4. Some printing happens, so that you can see a little bit of the data. Again, it's a *bad* idea to print all of the sound samples—some versions of Python can choke or freeze when printing so much data!
5. We already have the new sampling rate—that was the `newsr` argument. For consistency, we use the variable `newsamps` to label the old sound data samples. In this case they're not changing at all, but in later programs they will.
6. The code then writes `newsamps` and `newsr` out to a file, named `out.wav`, which will appear in the folder you're working in (replacing the old one if there was already an`out.wav`).
7. To finish, the function plays that new file, which may be a different speed than the original depending on the value of `newsr`.

The next example will illustrate how to create a new sound by changing the samples themselves.

Sound example #2: `flipflop`

This function should already be in your file, but if not, it's also here for easy reference and copy-and-paste:

```
def flipflop(filename):
    """ flipflop swaps the halves of an audio file
        input: filename, the name of the original file
        output: no return value, but
                this creates the sound file 'out.wav'
                and plays it
    """
    print( "Playing the original sound...")
    play(filename)

    print( "Reading in the sound data...")
    sound_data = [0,0]
    read_wav(filename,sound_data)
    samps = sound_data[0]
    sr = sound_data[1]

    print( "Computing new sound...")
    # this gets the midpoint and calls it x
    x = len(samps)//2
    newsamps = samps[x:] + samps[:x]
    newsr = sr
    new_sound_data = [ newsamps, newsr ]

    print( "Writing out the new sound data...")
    write_wav( new_sound_data, "out.wav" ) # write data to
out.wav

    print( "Playing new sound...")
    play( 'out.wav' )
```

Take a look at the middle part of this code, where the new sound samples are created from the old ones. In this case, the `newsamps` are a "flipflopped" version of the old `samps`.

As a result, the sound's second half is placed before its first half.

In building your audio-processing functions, use `flipflop` as a starting point.

Sound function to write #1: `reverse`

Next, write a sound-handling function `reverse` with the following signature:

```
def reverse(filename):
```

such that `reverse` takes in a `filename` as did `flipflop`.

Copy-and-paste `flipflop` to get started!

Like `flipflop`, the sampling rate should not change, but the function should create a *reversed* set of sound samples and then handle them in the same way as the two examples above. That is, you'll want to write them to the file `out.wav` and then play that file.

Remember to reverse the list `samps`, you can write `samps[::-1]` in Python!

Example(s):

```
In [1]: reverse('swfaith.wav')  # redaV htraD sounds eerier but
less intimidating
... lots of printing ...
```

Note that this `reverse` function ***won't*** need to use one any the helper functions you wrote above, but the next few will!

Sound function to write #2: `volume`

Now, write a sound-handling function `volume` with the following signature:

```
def volume(filename, scale_factor):
```

such that `volume` takes in a `filename` as usual and a floating-point value `scale_factor`. Then, `volume` should handle the sound in the usual way, with the output file and played sound being scaled in amplitude (volume) by the scaling factor `scale_factor`. That is, each sample should be multiplied by `scale_factor`.

Here, you could use the helper function `scale` you wrote at the beginning of the lab... .

Example(s):

```
In [1]: volume( 'swfaith.wav', .1 )  # A calmer Darth...
... lots of printing ...

In [2]: volume( 'swfaith.wav', 10.0 )  # A caffeinated Darth!
... lots of printing ...
```

You'll notice that your hearing adjusts remarkably well to this function's changes in absolute volume, making the perceived effect considerably less than you might expect.

You will also find that if you *increase* the volume too much, the sound becomes distorted, just as when an amplifier is turned up to 11.

Sound function to write #3: `static`

Now, write a sound-handling function `static` with the following signature:

```
def static(filename, probability_of_static):
```

such that `static` takes in a `filename` as usual and a floating-point value `probability_of_static`, which you can assume will be between 0 and 1.

Then, `static` should handle the sound in the usual way, with the output samples being replaced with the probability of `probability_of_static`. When they're replaced, the samples should simply be random values, uniformly chosen in the valid range from -32768 to 32767.

Here, you should use the helper function `replace_some` that you wrote earlier in the lab. You won't need `randomize`, because `replace_some` already uses it!

Example(s):

```
In [1]: static('swfaith.wav', .05)  # Vader, driving into a
tunnel
... lots of printing ...

In [2]: static('swfaith.wav', .25)  # Vader on dial-up from a
galaxy far, far away
```

```
... lots of printing ...
```

You might see how high you can increase the percentage of static until the original is no longer discernable. People adapt less well to this than to volume changes.

Sound function to write #4: `overlay`

Now, write a sound-handling function `overlay` with the following signature:

```
def overlay(filename1, filename2):
```

such that `overlay` takes in *two* `filenames` as usual, and it creates a new sound that overlays the two. The result should be as long as the shorter of the two. (Drop any extra samples, just as in `add_scale_2`.)

Use your `add_scale_2` helper function to assist with this! That way, you can adjust the relative loudness of the two input files. You are welcome, but certainly not required, to add more input arguments to your `overlay` function so that you can change the relative volumes on the fly (or crop the sounds on the fly, which is a bit more ambitious).

Remember that `add_scale_2(samps1, samps2, 0.5, 0.5)` ***must*** take lists (`samps`) as input—not filenames, which are simply strings! The `samps` are lists of the raw sound data.

Example(s):

```
In [1]: overlay( 'swfaith.wav', 'swnotry.wav' )  # Vader vs.
Yoda
... lots of printing ...
```

The next function overlays a file with a shifted version of itself.

Sound function to write #5: `echo`

This one is more of a challenge... .

Try writing a sound-handling function `echo` with the following signature:

```
def echo(filename, time_delay):
```

such that `echo` takes in a `filename` as usual and a floating-point value `time_delay`, which represents a number of seconds.

Then, `echo` should handle the sound in the usual way, with the original sound being overlaid by a copy of itself shifted forward in time by `time_delay`.

To do the overlaying, you'll want to use `add_scale_2`, as before.

To handle the time-shifting, notice that you can use the sampling rate to convert between the number of samples and time in seconds:

- For example, if `time_delay` is `0.1` and the sampling rate is `22050`, then the number of samples to wait is `2205`
- Similarly, if `time_delay` is `0.25` and the sampling rate is `44100`, then the number of samples to wait is `11025`

Hint on how to "add wait time" to samples...: the easiest way to add "blank space" or "blank sound" in front of `samps` is to concatenate (add a list of) zeros to the front of the list `samps`. For example,

```
samps2 = [0]*42 + samps
```

would "wait" 42 samples, by including 42 blank-sound samples, at the start of the sound data `samps`.

You'll probably want a value other than 42 - in fact, the challenge is to compute the *correct* value there!

How could you figure out what integer you need *instead of 42* ... ?

- remember that you know the time you'd like (in seconds) and the sampling rate (in samples per second) ...
- Be sure that you use an integer - remember that, if you have a floating-point value `f`, then `int(f)` is an integer.
- By the way, there are other approaches that work for `echo`, as well! with thanks to Sophie Harris for inventing one of the alternatives!

Example(s):

```
In [1]: echo( 'swfaith.wav', .1 )  # How many zeros would be
needed in front?
... lots of printing ...
```

Sound Example #3: generating pure tones

The final examples of provided functions generate a pure sine-wave tone. Here is the code, though it should also be in the file:

```
# Helper function for generating pure tones
def gen_pure_tone(freq, seconds, sound_data):
    """ pure_tone returns the y-values of a cosine wave
        whose frequency is cyclespersec hertz
        it returns nsamples values, taken once every 1/44100 of
a second
        thus, the sampling rate is 44100 hertz
        0.5 second (22050 samples) is probably enough
    """
    if sound_data != [0,0]:
        print("Please input a value of [0,0] for sound_data.")
        return
    sampling_rate = 22050
    # how many data samples to create
    nsamples = int(seconds*sampling_rate) # rounds down
    # our frequency-scaling coefficient, f
    f = 2*math.pi/sampling_rate   # converts from samples to Hz
    # our amplitude-scaling coefficient, a
    a = 32767.0
    sound_data[0] = [ a*math.sin(f*n*freq) for n in
range(nsamples) ]
    sound_data[1] = sampling_rate
    return sound_data


def pure_tone(freq, time_in_seconds):
    """ swaps the 2nd half with the 1st half """
    print("Generating tone...")
    sound_data = [0,0]
    gen_pure_tone(freq, time_in_seconds, sound_data)

    print("Writing out the sound data...")
    write_wav( sound_data, "out.wav" ) # write data to out.wav

    print("Playing new sound...")
    play( 'out.wav' )
```

Look over this code and try it out to get a feel for what it does, though the math of the sine wave is not crucial.

Rather, the important details are that the function `pure_tone` takes a desired frequency `freq` and the span `time_in_seconds`. The mathematical details are then delegated to `gen_pure_tone`.

Example(s):

```
In [1]: pure_tone(440, 0.5)  # 0.5 seconds of the concert-tuning
A
... lots of printing ...
```

You can look up frequencies for other notes at [at this Wikipedia page](), among many others. Here's a small chart, as well:

Below is a table of pitch frequencies in equal temperament, based on A4 = 440 Hz to the nearest Hertz (middle C = C4).

0	1	2	3	4	5	6	7	8
C 16	C 33	C 65	C 131	C 262	C 523	C 1047	C 2093	C 4186
C# 17	C# 35	C# 69	C# 139	C# 278	C# 554	C# 1109	C# 2218	C# 4435
D 18	D 37	D 73	D 147	D 294	D 587	D 1175	D 2349	D 4699
D# 20	D# 39	D# 78	D# 156	D# 311	D# 622	D# 1245	D# 2489	D# 4978
E 21	E 41	E 82	E 165	E 330	E 659	E 1319	E 2637	E 5274
F 22	F 44	F 87	F 175	F 349	F 699	F 1397	F 2794	F 5588
F# 23	F# 46	F# 93	F# 185	F# 370	F# 740	F# 1475	F# 2960	F# 5920
G 25	G 49	G 98	G 196	G 392	G 784	G 1568	G 3136	G 6272
G# 26	G# 52	G# 104	G# 208	G# 415	G# 831	G# 1661	G# 3322	G# 6645
A 28	A 55	A 110	A 220	A 440	A 880	A 1760	A 3520	A 7040
A# 29	A# 58	A# 117	A# 233	A# 466	A# 932	A# 1865	A# 3729	A# 7459
B 31	B 62	B 124	B 247	B 494	B 988	B 1976	B 3951	B 7902

Sound function to write #6: `chord`

The final lab problem is to build on the above example to write a chord-creation function named `chord` with the following signature:

```
def chord(f1, f2, f3, time_in_seconds):
```

such that `chord` takes in three floating-point frequencies `f1`, `f2`, and `f3`, along with a floating-point `time_in_seconds`. Then, `chord` should create and play a three-note chord from those frequencies.

You will want to get ***three*** sets of `samps` and `sr` from `gen_pure_tone`, e.g.,

```
samps1, sr1 = gen_pure_tone( f1, time_in_seconds, [0,0] )
samps2, sr2 = gen_pure_tone( f2, time_in_seconds, [0,0] )
samps3, sr3 = gen_pure_tone( f3, time_in_seconds, [0,0] )
```

Here, you really need an `add_scale_3` function, though we don't have that yet. But you can create it! (You could use `add_scale2` and `add_3` as starting points, but we'd recommend writing `add_scale_3` on its own—not calling those other functions.)

Example(s):

```
In [1]: chord(440.000, 523.251, 659.255, 1.0)   # A minor chord
... lots of printing ...
```

If your chord sounds static-y, you may have used `add_scale_3` with ***too large*** a set of coefficients for the combined sounds!

- You'll want to keep the overall amplitude at `1.0`
- Since the original amplitude is `1.0`, you'll need to use fractional scale values to make sure the overall amplitude of the summed waves stays at `1.0` or less
- If the wave exceeds `1.0` in amplitude, it will be "clipped" by the speakers, which then sounds like loud static overlaying the sound... .

Challenge: Use the table of frequencies above to change that chord from an A-minor to an A-major chord... . Or build your own...

Congratulations! For this lab, you're ready to submit your `hw3pr1.py` code. However, you ***don't*** need to submit any of the other files than `hw3pr1.py`. ***Be sure to rename the file `hw3pr1.py` before submitting it!***

Here is a link to the submissions site.

But what about creating a C minor 7th (augmented) chord?

Indeed, you might want to create larger chords with arbitrarily many notes ... or other unusual/odd/interesting/inspired/disturbing algorithmically-generated sound effects. We certainly encourage you to try things out!

Alternatively, you might look at the next problems on the appropriate assignment page:

- Homework3Gold
- Homework3Black

Or, you can simply bring down the curtain on this lab and head out towards an entirely different encore!

Either way, be sure to submit your `hw3pr1` file by next Monday evening... !

Full starter file for reference

```
# CS5 Gold, Lab 3
# Filename: hw3pr1.py
# Name:
# Problem description: Lab 3 problem, "Sounds Good!"

import time
import random
import math
import csaudio
from csaudio import *
import wave
wave.big_endian = 0  # needed in 2015
# if you are having trouble, comment out the above line...



# a function to get started with a reminder
# about list comprehensions...
def three_ize( L ):
    """ three_ize is the motto of the green CS 5 alien.
        It's also a function that takes in a list and
        returns a list of elements each three times as large.
    """
    # this is an example of a list comprehension
    LC = [ 3 * x for x in L ]
    return LC


# Function to write #1:  scale




# here is an example of a different method
# for writing the three_ize function:
def three_ize_by_index( L ):
    """ three_ize_by_index has the same I/O behavior as three_ize
```

```
        but it uses the INDEX of each element, instead of
        using the elements themselves -- this is much more flexible!
    """
    # we get the length of L first, in order to use it in range:
    N = len(L)
    LC = [ 3 * L[i] for i in range(N) ]
    return LC


# Function to write #2:  add_2




# Function to write #3:  add_3




# Function to write #4:  add_scale_2







# Helper function:  randomize

def randomize( x, chance_of_replacing ):
    """ randomize takes in an original value, x
        and a fraction named chance_of_replacing.

        With the "chance_of_replacing" chance, it
        should return a random float from -32767 to 32767.

        Otherwise, it should return x (not replacing it).
    """
    r = random.uniform(0,1)
    if r < chance_of_replacing:
        return random.uniform(-32768,32767)
    else:
        return x






# Function to write #5:  replace_some









#
# below are functions that relate to sound-processing ...
#


# a function to make sure everything is working
def test():
    """ a test function that plays swfaith.wav
        You'll need swfailt.wav in this folder.
```

```
    """
    play( 'swfaith.wav' )


# The example changeSpeed function
def changeSpeed(filename, newsr):
    """ changeSpeed allows the user to change an audio file's speed
        input: filename, the name of the original file
               newsr, the new sampling rate in samples per second
        output: no return value, but
                this creates the sound file 'out.wav'
                and plays it
    """
    print("Playing the original sound...")
    play(filename)

    sound_data = [0,0]            # an "empty" list
    read_wav(filename,sound_data) # get data INTO sound_data

    samps = sound_data[0]         # the raw pressure samples

    print( "The first 10 sound-pressure samples are\n", samps[:10])
    sr = sound_data[1]            # the sampling rate, sr

    print( "The number of samples per second is", sr)

    # we don't really need this line, but for consistency...
    newsamps = samps                        # same samples as before
    new_sound_data = [ newsamps, newsr ]   # new sound data pair
    write_wav( new_sound_data, "out.wav" ) # write data to out.wav
    print("\nPlaying new sound...")
    play( 'out.wav' )   # play the new file, 'out.wav'



def flipflop(filename):
    """ flipflop swaps the halves of an audio file
        input: filename, the name of the original file
        output: no return value, but
                this creates the sound file 'out.wav'
                and plays it
    """
    print( "Playing the original sound...")
    play(filename)

    print( "Reading in the sound data...")
    sound_data = [0,0]
    read_wav(filename,sound_data)
    samps = sound_data[0]
    sr = sound_data[1]

    print( "Computing new sound...")
    # this gets the midpoint and calls it x
    x = len(samps)//2
    newsamps = samps[x:] + samps[:x]
    newsr = sr
    new_sound_data = [ newsamps, newsr ]

    print( "Writing out the new sound data...")
    write_wav( new_sound_data, "out.wav" ) # write data to out.wav

    print( "Playing new sound...")
    play( 'out.wav' )



# Sound function to write #1:  reverse




# Sound function to write #2:  volume
```

```
# Sound function to write #3:  static




# Sound function to write #4:  overlay




# Sound function to write #5:  echo




# Helper function for generating pure tones
def gen_pure_tone(freq, seconds, sound_data):
    """ pure_tone returns the y-values of a cosine wave
        whose frequency is cyclespersec hertz
        it returns nsamples values, taken once every 1/44100 of a second
        thus, the sampling rate is 44100 hertz
        0.5 second (22050 samples) is probably enough
    """
    if sound_data != [0,0]:
        print("Please input a value of [0,0] for sound_data.")
        return
    sampling_rate = 22050
    # how many data samples to create
    nsamples = int(seconds*sampling_rate) # rounds down
    # our frequency-scaling coefficient, f
    f = 2*math.pi/sampling_rate   # converts from samples to Hz
    # our amplitude-scaling coefficient, a
    a = 32767.0
    sound_data[0] = [ a*math.sin(f*n*freq) for n in range(nsamples) ]
    sound_data[1] = sampling_rate
    return sound_data


def pure_tone(freq, time_in_seconds):
    """ swaps the 2nd half with the 1st half """
    print("Generating tone...")
    sound_data = [0,0]
    gen_pure_tone(freq, time_in_seconds, sound_data)

    print("Writing out the sound data...")
    write_wav( sound_data, "out.wav" ) # write data to out.wav

    print("Playing new sound...")
    play( 'out.wav' )



# Sound function to write #6:  chord
```