CS 112 – Project 5 Dictionaries, File I/O, and Modules

Due Date: Tuesday, Nov 26th, 11:59pm

early EC deadline: Sunday, Nov 24th, 11:59PM

Project basics file:

- https://cs.gmu.edu/~marks/112/projects/project_basics.pdf

Needed file. A zip file containing all the tester files

- https://cs.gmu.edu/~marks/112/projects/P5.zip

Background

Dictionaries give an enriched way to store values by more than just sequential indexes (as lists give us); we identify key-value pairs, and treat keys like indexes of various other types. The only restriction on keys is that they are "hashable", which we can approximate by thinking that they are "immutable all the way down". Though unordered, dictionaries help us simplify many tasks by keeping those key-value associations. Each key can only be paired with one value at a time in a dictionary.

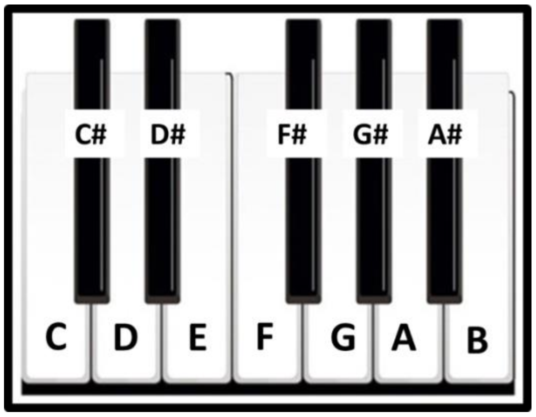
When a file contains text, we can readily write programs to open the file and compute things based on the file's contents. This also gives our programs far more longevity: we can store data and results for later, save user preferences, and all sorts of other useful things.

Scenario

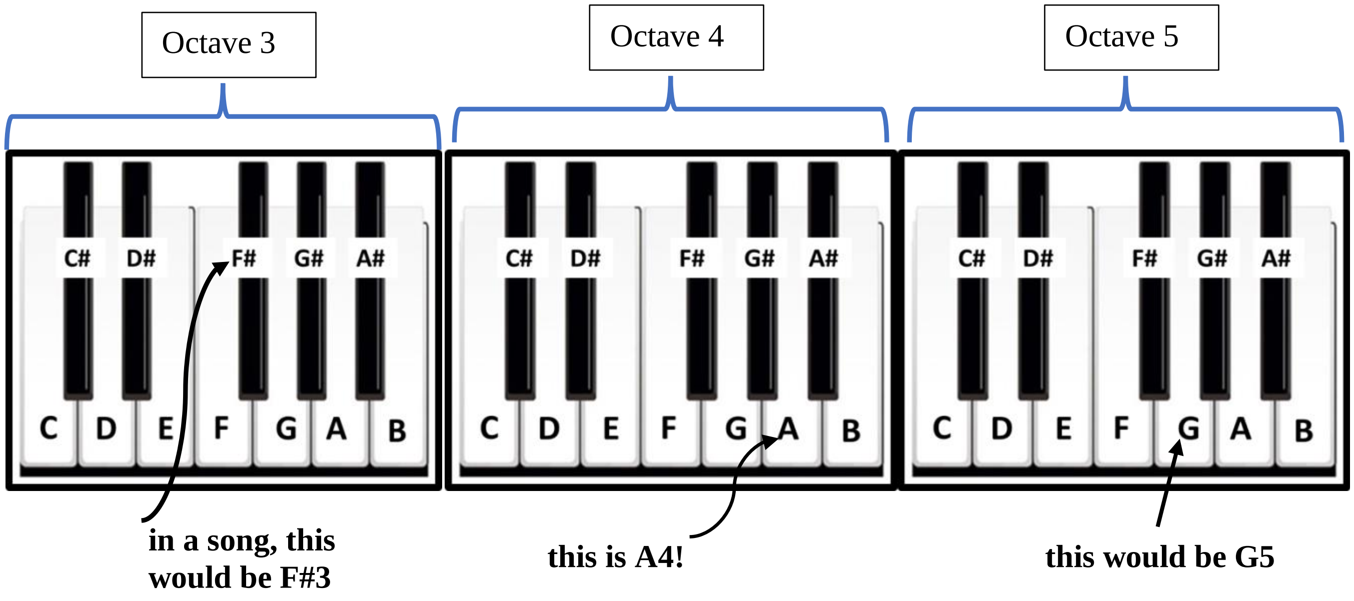
In this project you will get practice creating and manipulating dictionaries, and reading/writing files, by working with special "song" files that contain all the information needed to play a song on a piano (and subsequently on your computer.)

Definitions

There are 7 basic notes on a piano each given the letter A, B, C, D, E, F, or G. All of these notes are played by pressing one of the white keys. There are notes that exist in between some of the standard notes, which are referred to as "sharps." They are played by pressing the black keys on a piano. One "octave" on the piano is the group of 12 notes from C to B, pictured below. Since there are sharp notes, we consider each of the keys that are next to each other as one 'half-step' apart. This means that C to C# is one half step, but C to D would be two half steps. Notice how there is no extra sharp note between E and F, or B and C. Both of these adjacent pairs of notes are still considered one half-step apart.



A standard piano has 88 keys that just repeat this pattern of notes over and over again. As you travel to the right the notes get higher, and they get lower as your travel to the left. The notes that are in higher octaves sound higher pitched, and the lower octaves sound lower pitched. You will use a combination of the note's letter and the octave it is played in to accurately describe an individual key on the piano.



Notice how the notes themselves are repeated several times. A standard piano has 7 full octaves (1 through 7), but the first key will actually start at A0, and the last key will be C8. Octaves 0 and 8 on the piano are therefore incomplete, they will only have the notes A0, A#0, B0, and C8.

By pressing any of the keys on the piano, you are causing a small hammer to hit a string that will vibrate at a particular frequency. A note is octave 5 will have twice the frequency of the same note in octave 4, which is why the octave 5 note sounds higher pitched. You can compute the frequencies for each note by implementing the following formula:

$$F_n = F_0 \times 2^{n/12}$$

F_0 is the frequency of base note, in other words the note that you are using to tune all of the other notes. It is standard to tune your piano so that the note A4 is exactly 440Hz, but the function you write to generate the frequencies of the notes will use any arbitrary starting frequency.

n is the number of half steps you are away from the base note (A4):

- n > 0 means you are on a note above the base note
  - A#4 is one half step above A4, so n=1
  - B4 is two half steps above A4, so n=2
  - C#5 is four half steps above A4, so n=4
  - A5 is twelve half steps above A4, so n=12
- n < 0 means you are on a note below the base note
  - G#4 is one half step below A4, so n = -1
  - G4 is two half steps below A4, so n = -2
  - A3 is twelve half steps below A4, so n = -12
  - F3 is sixteen half steps below A4, so n = -16

Note Types

There are few different speeds at which a note can be played in a piece of music. We generally think of this in terms of 'beats' rather than seconds or minutes. A song will describe its overall tempo in beats per minute (BPM), where a higher BPM means a faster song. When reading music, the notes are organized in "measures." The composer of the song would tell you how many beats should be in one measure. We will assume that all music used in this project uses 4 beats per measure. This makes computing the speeds easier.

We have five different note types in this project. The name of the note type indicates how much of one measure that note should occupy:

- "Quarter" notes take up a quarter of the measure, so for us that is exactly 1 beat
- "Half" notes take up half of the measure, that would mean they last for 2 beats in our songs
- "Whole" notes take up the entire measure, so they require 4 beats to play correctly
- "Eighth" notes take up an eighth of the measure, so that would be half of one beat
- "Sixteenth" notes take up a sixteenth of a measure, so that would be a quarter of one beat

Song Files

Files that contain songs in this project are denoted by the extension ".song", and are organized like this:

```
SongTempoInBPM
BaseNoteFrequency
NoteLetterAndOctave,NoteType
NoteLetterAndOctave,NoteType
```

The first line of the file is the song's tempo expressed in beats per minute.
The second line of the file is frequency of the base note used to tune the rest of the notes.
The rest of the lines contain the notes in the order that they should be played for a particular song. They are comma separated, where the first item is the note's letter on the piano and octave that is should be played in.

You can open up these files in any plain text editor, they use only ASCII characters.

Restrictions

- There is exactly one `import` statement allowed in this project: `import random`
    - You **are not** allowed to `import` anything else.
    - You can only use `randint()` `choice()` and `seed()` from the random library
- You **are not** allowed to use classes, exceptions, the with statement, and list/dictionary comprehensions.
- From the built-in functions, you ***are*** allowed to use:
    - `range(), len(), sorted(), int(), sum(), list(), float()`
- From list methods, you ***are*** allowed to use:
    - `append(), insert(), copy(), remove(), pop(), index()`
- From the dictionary methods, you **are** allowed to us:
    - `get(), values(), keys(), items(), update()`
- For the file/string methods, you are allowed to use:
    - `open(), strip(), split(), join()`
    - `read()`, `readline(), readlines()`, `write(), writelines()`
- You may not make more than one pass of a file when reading it. **<u>Do not circumvent this rule</u>** by reading in the file as one entity (one string, a list, etc.), and then performing multiple passes on that.
- Questions on Piazza like "Can I use [thing not listed above]?" will not be answered by an instructor.

Assumptions

You may assume that:

- The types of the function arguments are the proper ones, you don't have to validate them.

Testing

All the necessary files for testing your project have been **zipped** in one file. Once you download it, you must unzip it, and then work in it without moving or renaming its contents, otherwise the tester will not work. Tester inputs and outputs are stored in separate sub-directories, do not move or rename them either. It is recommended that you **test each function independently** because some tests might be computationally intensive.

Other than that, this tester is still very similar to the previous ones. Be reminded that some of the test cases we'll be using for grading are omitted from the tester. This means that there might be errors in your code even if the tester is giving you no errors. You must do your own checks to make sure that you haven't missed anything and your code is correct. You do ***not*** need to modify the tester, just test on your own any way you like.

Grading Rubric

Correct submission & adequate commenting:	10	# see project basics file for more info
Tester cases:	75	# see project basics file for how to test!
Unknown test cases:	15	# you have to manually do your own checks
Extra credit:	5	# early submission
TOTAL:	105	

Functions

The following are the functions you must implement. The signature of each one is provided, do ***not*** make any changes to them otherwise the tester will not work properly. Keep in mind that you do ***not*** have to write a main body for your program in this project. There is a small starter file provided that already imports the random module for you.

`generate_durations(base_tempo)`

Description: Generates a dictionary of note durations dictated by the base tempo. This is how long each note should be played based on tempo of the song. Assume that one measure of the song is equivalent to four beats. A quarter note would then last for one beat, a half note for two beats, a whole note for four beats, an eighth note for half of a beat, and sixteenth note for a quarter of a beat.

Parameters: **`base_tempo`** (positive integer), expressed in beats per minute.

Return value: a dictionary of keys (strings) and values (floats) that maps note types to seconds

Example:

```
generate_durations(60) --> {'Whole': 4.0, 'Half': 2.0, 'Quarter': 1.0,
                            'Eighth': 0.5, 'Sixteenth': 0.25}
```

`generate_frequencies(base_freq)`

Description: Generates a dictionary of frequency values for all eight octaves on the piano. See the above description of the problem for a detailed look at the formula used to compute these frequencies. The lowest note you should generate a frequency for is A0, and the highest note is C8, and every note in between those two.

Parameters: **`base_freq`** (positive float) the frequency to tune the note 'A4' to

Return value: a dictionary of notes (strings) and corresponding frequencies (floats)

Example:

```
generate_frequencies(440)   ->   {'A0': 27.5, […more notes…]
                                  'A4': 440.0, […more notes…]
                                  'C8': 4186.009044809578}
```

`find_note(filename, highest_or_lowest)`

Description: Determine the highest or the lowest note found in the song. You can assume that all notes in the song will be ones found on an 88 key piano.

Parameters: **`filename`** (string) the name of the file containing the song, **`highest_or_lowest`** (Boolean) is a flag that controls whether you are finding the lowest or the highest note.

Return value: a string: the highest note in the song if **`highest_or_lowest`** is set to **`True`**, the lowest note if it is set to **`False`**

Example:

```
find_note("test.song", True)    ->   "B5"
find_note("test.song", False)   ->   "C2"
```

"test.song"

```
90
440
A4,Quarter
B5,Half
C2,Quarter
D#2,Whole
```

`random_song(filename, tempo, tuning, num_measures)`

Description: Uses the random module to write song made of random notes to a file given the name of the song to write to, the desired tempo, and the desired tuning frequency, and the number of measures this song is. We want to make sure that the song you generate would be a valid song in terms of beats in measures, so you are going to pick random notes using the following algorithm:

1. Loop as long as you have not completed the number of measures that is being asked for, then build the next measure by doing the following:
    a. Pick a random valid index in **this exact list:**
    `["Sixteenth", "Eighth", 'Quarter', 'Half','Whole']`
    b. If this note type at the index that you pick can fit in this measure (the number of beats it requires is less than or equal to the number of beats left in the measure) then:
        i. Select a random index in **this exact list:**
        `["C","C#","D","D#","E","F","F#","G","G#","A","A#","B"]`
        ii. Select a random octave from 1 to 7 (we're ignoring the incomplete octaves 0 and 8)
        iii. join the note type, note, and octave together, and write this as a line in the output file
    c. If the note duration selected doesn't fit in the measure, throw away this note and try again in the next iteration of the loop, to pick a new random index to try until a note type that fits is selected.

Parameters: `filename` (string) the name of the output file to write to, `tempo` (int) the BPM of the song, `tuning` (float) the base frequency to use for the note A4, and `num_measures` (int) the number of measure of song to write

Return value: None (all output goes to a file)

Example: *The output is random, so for examples of this it is best to check the provided files.*

Random module: you should import the random module: `import random`

Review these methods (test them by themselves, perhaps in the interpreter!) as options for selecting values (see the docs here: https://docs.python.org/3/library/random.html)

- **`random.randint(low_inclusive, high_inclusive)`** , useful for selecting a number in your indicated range with equal probability. Note that the upper and lower values are inclusive.
- **`random.choice(sequence)`** , which selects an item from a sequence with equal probability and returns it.
- **`random.seed(int)`**, which the tester uses to "lock in" the sequence that this pseudo-random number generator will generate. Note that the exact number of calls to the random module also affect what values are chosen, so read the directions carefully! It should exactly clarify how often, and in what order, to make calls to the random module. When running your own individual tests **you must make sure that your seed value matches the seed value of the test you're using**, otherwise you will never be able to match the output and you'll be sad.

`change_notes(filename, changes, shift)`

Description: Changes the notes of the song given by the filename based on the parameters passed to the function. **If a note appears** as a key in the dictionary of changes, it should just be changed to the corresponding value. **Otherwise**, the note should be shifted up or down the piano by **`shift`** number of half steps on the piano. You should not change notes in the song that would be shifted past a valid key on the piano on the left or on the right. For example, C8 should remain C8 if the shift value was 4. The resulting song should be written to a file, given the name of the original file "_changed" added before the extension ".song"

Parameters: **`filename`** (string) the name of the file containing the song to change, **`changes`** is a dictionary that maps notes to other notes, it won't necessarily have every note in it, **`shift`** (integer) is how many half steps up or down the piano from the original note you should travel in order to find the replacement note in the changed file

Return value: None (the changed song should be written to a new file)

Example:

"test.song"

```
90
440
A4,Quarter
A#2,Quarter
B5,Quarter
C2,Quarter
C#2,Quarter
D3,Quarter
D#2,Quarter
E6,Quarter
F7,Quarter
F#1,Quarter
G5,Quarter
G#6,Quarter
```

using the song on the left, the changed song is on the right

```
change_notes('test.song', {'C2':'E3'}, 5)
```

"test_changed.song"

```
90
440
D5,Quarter
D#3,Quarter
E6,Quarter
E3,Quarter
F#2,Quarter
G3,Quarter
G#2,Quarter
A6,Quarter
A#7,Quarter
B1,Quarter
C6,Quarter
C#7,Quarter
```

`song_as_dict(filename)`

Description: Transforms a song from a file into a dictionary. The information it stores is the number of notes (separated by octaves) and note types that appear in the given song.

Parameters: `filename` (string) the name of the song file to use

Return value: A dictionary with **four keys**: `"tempo"`, `"tuning"`, `"notes"` and `"types"`

The "tempo" key is just the value of the tempo of the song, and the "tuning" key is the value of the tuning frequency of the song. The "notes" key is a dictionary that maps each individual note to another. dictionary of how many times the note appears per octave in the song. The "types" key is a dictionary that maps each note type to the number of times that note type appears in the song. Note that any notes not present in the song are also not present in the dictionary.

Example:

```
song_as_dict("test.song")   ---->

 {
 'tempo': 90,
 'tuning': 440.0,
 'notes': {  'D': {6: 1, 4: 1},
             'F#': {6: 2, 7: 2, 2: 1},
             'A#': {1: 1},
             'A': {2: 1},
             'D#': {2: 1},
             'E': {6: 1},
             'G': {5: 1},
             'G#': {5: 1},
             'C': {4: 1},
             'B': {2: 1}
             },
 'types': {  'Whole': 2,
             'Quarter': 1,
             'Half': 2,
             'Sixteenth': 8,
             'Eighth': 2
             }
 }
```

"test.song"

```
90
440
D6,Whole
F#6,Quarter
A#1,Half
A2,Sixteenth
D#2,Eighth
F#7,Sixteenth
F#7,Whole
D4,Sixteenth
E6,Sixteenth
G5,Eighth
G#5,Half
C4,Sixteenth
F#2,Sixteenth
B2,Sixteenth
F#6,Sixteenth
```