CSE 101: Computer Science Principles (Fall 2019)

Homework #2

Assignment Due: Tuesday, November 26, 2019, by 11:59 PM

Assignment Objectives

This assignment asks you to develop a series of functions that work together to read and process the data stored in a plain text file. You will extend this code in the final project, which will use these functions to build a larger, more powerful program.

Getting Started

This assignment requires you to write Python code to solve several computational problems. To help you get started, we will give you a basic starter file for each problem with a few tests you can use to verify that your code seems to be correct (**note that these test cases are just examples; we will use different test inputs to grade your work!**). For each problem, complete its function as directed in the assignment. **Do not, under any circumstances, change the names of the functions or their parameter lists.** The automated grading system will be looking for functions with those exact names and parameter lists; if you change any function headers, the grading program will reject your solution and mark it as incorrect.

Directions

Each part of this assignment describes a single function that makes up part of the final program. The starter code for these functions is specially designed to work together. Specifically, Parts II–V all use your solution from Part I in order to operate correctly. We have already set up the starter code files to correctly link everything together; **you MUST keep ALL of those .py files, as well as the sample data files, together in the same folder.**

As with the lab assignments, our automated grading system will execute your solution to each part several times, using different input values each time. Each test that produces the correct/expected output will earn 1 or more points. This assignment contains 5 problems, and is worth a total of 60 points. **Note that not every problem may be worth the same number of points!**

- ⚠ Each starter file has a comment block at the top with various important information. Be sure to add your information (name, ID number, and NetID) to the first three comment lines, so that we can easily identify your work. **If you leave this information out, you may not receive credit for your work!**
- ⚠ Submit your work as a set of individual files (one per problem). **DO NOT** zip or otherwise compress your files, and **DO NOT** place all of your work in a single file. **If you do so, you will receive a 0 for this assignment!**
- ⚠ Each of your functions **MUST** use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters) can't be graded by our automated grading system, and may receive a grading penalty (or may not be graded at all).
- ⚠ Each of your functions must explicitly ***return*** its final answer; the grading program ignores anything printed by your code. **DO NOT** use `input()` anywhere before the `if __name__ == "__main__":` statement in your files; every function will receive any data it needs from its parameters. Programs that crash will likely receive a failing grade, so test your code **thoroughly** before submitting it.
- ⚠ Blackboard will provide information on how to submit this assignment. You **MUST** submit your completed work as directed by the indicated due date and time. We will not accept (or grade) any work that is submitted after the due date and time, or that is submitted before you have signed the course's Academic Honesty agreement.
- ⚠ **ALL** of the solution code that you submit **MUST** be your own work! You may not receive assistance from or share any materials with anyone else, except for the instructor and the (current) course TAs.

Assignment: Creating Custom Musical Playlists (Part 1)

Suppose that you have a side business working as a DJ for parties and other events. Every event runs for a different length of time and has a different clientele, so you need a way to quickly generate a customized playlist of songs from your music library. In this assignment, you will begin to develop a program that will handle this task for you (or let you know that the task is impossible using the current music library and/or with the specified set of constraints); for the final project, you will finish this program by writing code to implement the remaining functions, as well as any "glue code" needed to connect the different components (functions) that you have written.

Part I: Assembling Your Music Library (15 points)

(Place your answer to this problem in the "createLibrary.py" file)

Function Summary

Parameters: A string that represents the name of a data file to open and read.

Return Type: A dictionary whose keys are strings. Each key's value is a list of four-element lists; each sublist represents a single song, and holds two strings and by two integers. If the file is empty, the function returns an empty dictionary.

Our first task is to assemble our library of songs. Each song has several attributes associated with it:

- the song title
- the artist who performed the song
- the song's genre — a one-word string like "rock" or "jazz"
- the song's length — an integer representing the song's total length in seconds
- the song's rating — an integer in the range 1–5, or 0 if the song has not been rated yet

We will collect this information from a (plain text) data file and store it in a dictionary. Each key in the dictionary represents a specific genre; its value will be a list of songs (from the data file) that belong to that genre. A song is represented by a four-element list whose elements appear in the following order: song title (a string), artist (a string), total length in seconds (an integer), and current rating (an integer).

For example:

```
"rock" : [ ["We're Not Gonna Take It", "Twisted Sister", 217, 5],
           ["Born To be Wild", "Steppenwolf", 211, 2],
           ["Pinball Wizard", "The Who", 182, 4] ]
```

indicates that our library contains three songs that belong to the "rock" genre.

Complete the `createLibrary()` function, which takes a single string argument (the name of a data file). `createLibrary()` returns a dictionary whose contents match the contents of the data file, in the format above. If the file is empty, the function returns the empty dictionary {}.

Each line of the data file contains information about a single song, in the following order:

song genre, song title, artist, song length (in MM:SS format), song rating

Fields are separated by the vertical bar (pipe) character | (assume that there are no leading/trailing spaces between fields).

For example, `jazz|Sputnik|Conrad Boqvist|5:05|2` describes a jazz song named "Sputnik" by Conrad Boqvist, whose length is 305 seconds (5 minutes, 5 seconds) with a 2-star rating.

Hint: To convert the time string into seconds, split it based on the colon (":") character. Convert each piece to an integer, and perform the appropriate arithmetic.

You may assume that the song length field always contains valid numbers for the time; you will never see a length value like -2:45 or 3:76. Likewise, you may assume that the song rating always falls in the range 0–5, inclusive.

In (very general) pseudocode:

- For each line in the data file:
  - Split the line based on pipe characters
  - Convert the "time" element into an integer number of seconds
  - Create a new list that holds the song's information (except for its genre) *in the same order as it appears in the data file*
  - Add the new list to your dictionary, using the genre as the key

Examples:

Here are examples of what `createLibrary()` should return for each of the sample data files that we provided with this assignment (the formatting may vary somewhat). Bear in mind that the ordering of key-value pairs in a dictionary does not necessarily match the order in which those values were added to the dictionary, so your answers may vary slightly from the ones below. Even so, you will receive full credit for this problem as long as your dictionary's contents are identical to those of the dictionary created by our reference solution, regardless of the specific order of the keys (or the order of the songs in each genre list). Remember that we will use different data files when we actually test and grade your code.

`createLibrary("library1.txt")` returns the dictionary:

```
{"childrens": [["Be Our Guest", "Angela Lansbury", 224, 0],
 ["Lullabye", "Billy Joel", 213, 0]], "dance": [["Happy Now", "Kygo", 211, 0],
 ["Grapevine", "Tiesto", 150, 0], ["Headspace", "Dee Montero", 271, 0]],
 "blues": [["Dream of Nothing", "Bob Margolin", 208, 0],
 ["Rock and Stick", "Boz Scaggs", 290, 0], ["At Last", "Etta James", 181, 0],
 ["You're Driving Me Crazy", "Van Morrison", 286, 0]],
 "kpop": [["Not That Type", "gugudan", 191, 0], ["IDOL", "BTS", 222, 0],
 ["Believe Me", "Seo In Young", 191, 0], ["Baam", "MOMOLAND", 208, 0],
 ["Hide Out", "Sultan of the Disco", 257, 0]]}
```

`createLibrary("library2.txt")` returns the dictionary:

```
{"jazz": [["Frankenstein", "Christian Sands", 523, 0],
 ["Lebroba", "Andrew Cyrille", 344, 0]],
 "classical": [["Perfect", "The Piano Guys", 310, 0],
 ["To Where You Are", "Katherine Jenkins", 226, 0]],
 "country": [["Chrome", "Trace Adkins", 203, 0],
 ["Nothing I Can Do About It Now", "Willie Nelson", 198, 0],
 ["Should've Been a Cowboy", "Toby Keith", 302, 0],
 ["Better Than You", "Terri Clark", 242, 0]]}
```

`createLibrary("library3.txt")` returns the dictionary:

```
{"metal": [["Call The Man", "Pentagram", 229, 0], ["Brimstone", "Whitechapel", 205, 0],
 ["Lightning Strike", "Judas Priest", 209, 0], ["Enter Sandman", "Metallica", 331, 0]],
 "reggae": [["Sidung", "Kranium", 200, 0], ["Naked Truth", "Sean Paul", 215, 0],
 ["Red Red Wine", "UB40", 320, 0]],
 "hiphop": [["Spaceship Odyssey 2000", "Malik Yusef", 276, 0],
 ["It's Tricky", "Run-DMC", 183, 0], ["Cuttin Rhythms", "Tone Loc", 311, 0]]}
```

`createLibrary("library4.txt")` returns the dictionary:

```
{"electronic": [["Fast Life", "Jackson and His Computer Band", 308, 0],
 ["Stairway to Heaven", "Jana", 245, 0], ["Silikon", "Modeselektor", 226, 0]],
 "rock": [["School's Out", "Alice Cooper", 210, 0], ["My Reply", "The Ataris", 254, 0],
 ["Burning Bridges", "Bon Jovi", 164, 0], ["Escher's Staircase", "Lana Lane", 366, 0]],
 "alternative": [["Psycho Killer", "Talking Heads", 260, 0],
 ["So Far So Good", "Thornley", 202, 0], ["Nothing At All", "Wired All Wrong", 198, 0],
 ["Chocolate", "The 1975", 227, 0]]}
```

Note that single quotes used as apostrophes may be preceded by a backlash as an *escape character*, e.g. `\'`.

Part II: Finding Every Song by a Specific Artist (10 points)

(Place your answer to this problem in the "getSongsByArtist.py" file)

Function Summary

Parameters: The first parameter is a dictionary whose keys are musical genres (strings) and whose values are lists of four-element lists (each sublist represents a specific song). The second argument is a string that contains the name of an artist.

Return Type: This function returns a list of strings, where each string is a song title from the dictionary whose artist matches the second parameter (note that the search is case-sensitive). If no songs by the specified artist are present, the function returns an empty list.

Complete the `getSongsByArtist()` function, which takes two arguments: a dictionary of songs, organized by genre (basically, the kind of dictionary created by the `createLibrary()` function from Part I), followed by a string containing the name of a musical artist or group. The function returns a list of strings, where each string is the name of a song by the specified artist that can be found in the current music library (the dictionary argument). The final list must contain **EVERY** song in the library by the specified artist, in any order. **For the purposes of this function, the artist's name is case-sensitive:** "KISS" is **NOT** the same as "Kiss" or "kiss". If the artist is not present anywhere in the library, the function returns the empty list.

For example, if the dictionary `lib` contained the following key-value entry:

```
["metal": [["One", "Metallica", 446, 2], ["Freak on a Leash", "Korn", 255, 1],
 ["And Justice for All", "Metallica", 585, 1], ["Zombie", "Bad Wolves", 254, 2],
 ["We're Not Gonna Take It", "Twisted Sister", 219, 4],
 ["Youth Gone Wild", "Skid Row", 199, 5], ["Enter Sandman", "Metallica", 331, 5]]
]
```

(in this case, Metallica only appears in a single genre)

`getSongsByArtist(lib, "Metallica")` would return the list:

`["One", "And Justice for All", "Enter Sandman"].`

Note that an artist may appear in multiple genres, so you will need to examine every genre in the library in order to assemble a complete (and correct) list.

Examples:

Using the "library1.txt" file as a data source:

- `getSongsByArtist()` returns `["At Last"]` for the argument "Etta James"
- `getSongsByArtist()` returns `[]` for the argument "Metallica"

Part III: Finding Every Artist in a Given Genre (10 points)

(Place your answer to this problem in the "getArtistsByGenre.py" file)

Function Summary

Parameters: The first parameter is a dictionary whose keys are musical genres (strings) and whose values are lists of four-element lists (each sublist represents a specific song). The second argument is a string that contains the name of a musical genre.

Return Type: This function returns a list of strings, where each string is the name of an artist who has at least one song in the library that belongs to the specified genre. An artist should only appear once in the list, even if he/she has multiple songs in the library. If the genre is not present in the library, the function returns an empty list.

Complete the `getArtistsByGenre()` function, which takes two arguments: a dictionary of songs, organized by genre (basically, the kind of dictionary created by the `createLibrary()` function from Part I), followed by a string containing the name of a musical genre. If the specified genre is present in the dictionary, the function should return the list that contains exactly **ONE** copy of the name of each artist who has one or more songs in that genre (in other words, if Artist A has two songs in the specified genre, Artist A's name should only appear once in the final list). The elements (names) in the final list may appear in any order, as long as they are unique. If the genre is not present (as a key) in the library, the function returns the empty list.

For example, if the dictionary `lib` contained the following key-value entry:

```
["metal": [["One", "Metallica", 446, 2], ["Freak on a Leash", "Korn", 255, 1],
 ["And Justice for All", "Metallica", 585, 1], ["Zombie", "Bad Wolves", 254, 2],
 ["We're Not Gonna Take It", "Twisted Sister", 219, 4],
 ["Youth Gone Wild", "Skid Row", 199, 5], ["Enter Sandman", "Metallica", 331, 5]]
]
```

`getArtistsByGenre(lib, "metal")` would return the list:

`["Metallica", "Korn", "Bad Wolves", "Twisted Sister", "Skid Row"].`

Examples:

Using the "library1.txt" file as a data source:

- `getArtistsByGenre()` returns `["Bob Margolin", "Boz Scaggs", "Etta James", "Van Morrison"]` for the genre "blues"
- `getArtistsByGenre()` returns `[]` for the genre "hiphop"

Part IV: Getting a List of Song Titles by Genre (10 points)

(Place your answer to this problem in the "getSongsByGenre.py" file)

Function Summary

Parameters: The first parameter is a dictionary whose keys are musical genres (strings) and whose values are lists of four-element lists (each sublist represents a specific song). The second argument is a string that contains the name of a musical genre.

Return Type: This function returns a list of strings, where each string is the title of a song that belongs to the specified genre. The list should contain the title of **every** song in the specified genre. If the genre is not present in the library, the function returns an empty list.

Complete the `getSongsByGenre()` function, which takes two arguments: a dictionary of songs, organized by genre (basically, the kind of dictionary created by the `createLibrary()` function from Part I), followed by a string containing the name of a musical genre. If the specified genre is present in the dictionary, the function should return a list that contains the name/title of **EVERY** song in that genre (but **ONLY** the song titles, without any other information). The elements (song names) in the final list may appear in any order, as long as the list is complete. If the genre is not present (as a key) in the library, the function returns the empty list.

For example, if the dictionary `lib` contained the following key-value entry:

```
["metal": [["One", "Metallica", 446, 2], ["Freak on a Leash", "Korn", 255, 1],
 ["And Justice for All", "Metallica", 585, 1], ["Zombie", "Bad Wolves", 254, 2],
 ["We're Not Gonna Take It", "Twisted Sister", 219, 4],
 ["Youth Gone Wild", "Skid Row", 199, 5], ["Enter Sandman", "Metallica", 331, 5]]
]
```

`getSongsByGenre(lib, "metal")` would return the list:

```
["One", "Freak on a Leash", "And Justice for All", "Zombie", "We're not Gonna Take
It", "Youth Gone Wild", "Enter Sandman"].
```

Examples:

Using the "library1.txt" file as a data source:

- `getSongsByGenre()` returns `["Dream of Nothing", "Rock and Stick", "At Last", "You're Driving Me Crazy"]` for the genre "blues"
- `getSongsByGenre()` returns `[]` for the genre "hiphop"

Part V: Finding All Songs with a Certain Minimum Length (15 points)

(Place your answer to this problem in the "getSongsLongerThan.py" file)

Function Summary

Parameters: The first parameter is a dictionary whose keys are musical genres (strings) and whose values are lists of four-element lists (each sublist represents a specific song). The second argument is a string that contains a time value in MM:SS format.

Return Type: This function returns a list of strings that contains the title of every song in the library whose length is strictly greater than the time from the second parameter. *ONLY* those songs should be included in the list. If the library does not contain any songs with the required length, the function returns an empty list.

Complete the `getSongsLongerThan()` function, which takes two arguments: a dictionary of songs, organized by genre (basically, the kind of dictionary created by the `createLibrary()` function from Part I), followed by a string representing a length of time in MM:SS format. The function should return a list of strings where each string is the title of a song in the library whose length is **strictly greater than** the specified time. The elements (song names) in the final list may appear in any order, as long as the list includes **EVERY** applicable song in the library, regardless of genre, and **ONLY** those songs. If the library doesn't contain any songs that meet the length requirement, the function should return an empty list.

NOTE: Remember that the library stores songs with their lengths converted into seconds. You need to take this format difference into account when you compare the length of each song to the required minimum length.

For example, calling `getSongsLongerThan()` with "3:15" as the second argument would return a list of every song in the specified library whose length was 196 seconds or more ("3:15" translates to 195 seconds), and only those songs.

Examples:

Using the "library1.txt" file as a data source:

- `getSongsLongerThan()` returns `["Be Our Guest", "Headspace", "Rock and Stick", "You're Driving Me Crazy", "IDOL", "Hide Out"]` for the length "3:35"
- `getSongsLongerThan()` returns `[]` for the length "7:30"