PA05: Song Manager

University of Virginia, Department of Computer Science
CS 1112 - Intro to Programming
Fall 2024

Due by: 11:00pm on Wednesday, October 30, 2024

Contents

Objective: The objective of this assignment for students is to learn how to work with **lists**, the **nested list** data structure, and **tuples** for a practical application. By the end of the assignment, students should be able to:

- Navigate lists of lists
- Retrieve from, and modify data in, a single list
- Understand the similarities and differences between working with lists and tuples

1 Background: lists of lists

A list of lists is a list where each entry is a different list. If each component is a list of strings, for example, a list of those lists might look like the following:

```
foods_by_letter = [
    ['apple', 'almond', 'artichoke', 'asparagus'],
    ['basil', 'beef', 'bread', 'broccoli'],
    ['cashew', 'cherry', 'chickpea', 'chocolate']
]
```

Accesses to list elements can occur at "two levels". For example:

```
    print(foods_by_letter[0])
```

will print `['apple', 'almond', 'artichoke', 'asparagus']`, which is the entire element at index 0, *not* the string `'apple'`. The statement:

```
    print(foods_by_letter[0][0])
```

however, *will* print `'apple'` because this is the element: ***A)*** at index zero of the list which is ***B)*** at index zero of the entire list.

A list of lists is one example of a *nested data structure*, which is a data structure whose members are other data structures. More data structures offer more freedom in representing data: a list of lists is the way the song database is represented in this assignment, but it is entirely possible to represent the data in something like a dictionary with list elements or a similar setup.

2 Task

As a course staff, we listen to a lot of music. In fact, we listen to *so much* music that we have trouble keeping track of it! We want **your** help to manage a database of our favorite songs and retrieve relevant information about them.

In a file called `song_manager.py` (named exactly), write a small library of functions to interact with this song database. The functions will include "helper functions" (smaller functions) and more involved functions which the helper functions will *help* (make easier to implement by reducing the code you have to write).

The song database is a list of lists. Each list is composed of strings corresponding to a person's name as well as information about a favorite song of theirs. A short example of the database we will test your code with might look like the following:

```
database_list = [
    ['Paul', 'Sledgehammer', 'Peter Gabriel', 'So', 'Funk', '1986'],
    ['Sam', 'Never Gonna Give You Up', 'Rick Astley', \
        'Whenever You Need Somebody', 'Dance-pop', '1987'],
    ['Jordan', 'Brother Louie', 'Modern Talking', 'Ready for Romance', \
        'Dance-pop', '1986'],
    # more entries go here...
]
```

The format of each list in the "database" (list of lists) is, in order,

```
    [name, title, artist, album, genre, year]
```

To borrow from the full example above, an entry that is given as

```
    ['Jordan', 'Brother Louie', 'Modern Talking', 'Ready for Romance',
    'Dance-pop', '1986']
```

means that the person this entry corresponds to is `'Jordan'`, that their favorite song is titled `'Brother Louie'`, that the song is by artist `'Modern Talking'`, the song is from the album `'Ready for Romance'`, the song's genre is `'Dance-pop'`, and the song is from `'1986'`.

In a Python file named `song_manager.py`, write a small library of functions that will interact with a "song database" that is a *list of lists*. We'll have you write both "helper functions" (small functions to do specific tasks) and more involved functions that can be implemented using the helper functions.

For all functions, you **do not** need to worry about getting the full database list–you can use our small example in this document to test your code, but we will pass in our own database in the autograder. The format of each list in the database, which is described above, **is constant**. Be sure to also keep in mind the "two levels" of access for lists of lists as described in Section 1.

Finally, when writing your code solutions, don't forget to include multi-line docstring comments and in-line comments!

2.1 find_entry(database_list, person_name)

Given a string `person_name`, find the **list within** `database_list` whose first item (at index `0`) matches `person_name`, and return that list. If there is no list within `database_list` with a name that matches `person_name`, return nothing.

For testing purposes, you do not need to worry about the possibility that multiple entries have names which match `person_name`; it is okay to only return the first match.
Example usage: given:

```
database_list = [
    ['Angela', 'Wurli', 'Dominic Fike', 'What Could Possibly Go Wrong', \
        'Alternative', '2020'],
    ['Evelyn', "You're My Best Friend", 'Queen', 'A Night at the Opera', \
        'Rock', '1976']
]
```

Calling `find_entry(database_list, "Angela")` should return the *entire list*:

```
    ['Angela', 'Wurli', 'Dominic Fike', 'What Could Possibly Go Wrong', \
        'Alternative', '2020']
```

That is, *all data* for the favorite song of `'Angela'` should be returned, and no data for the favorite songs of any other people in the list (such as `'Evelyn'`) should be returned.

2.2 add_entry(database_list, person_name, song_info)

Given a string `person_name` and a list of other data `song_info`, make a list with the two components that matches the given format, then add that combined list to `database_list`. This function should **modify `database_list` in place** but **not return anything.**
Example usage: given

```
database_list = [
    ['Angela', 'Wurli', 'Dominic Fike', 'What Could Possibly Go Wrong', \
        'Alternative', '2020'],
    ['Evelyn', "You're My Best Friend", 'Queen', 'A Night at the Opera', \
        'Rock', '1976']
]
```

Calling:

```
add_entry(database_list, "Paul", ['Sledgehammer', 'Peter Gabriel', 'So', \
    'Funk', '1986'])
```

should update `database_list` so that it now contains the lists (*in some order*):

```
    ['Angela', 'Wurli', 'Dominic Fike', 'What Could Possibly Go Wrong', \
        'Alternative', '2020'],
    ['Evelyn', "You're My Best Friend", 'Queen', 'A Night at the Opera', \
        'Rock', '1976'],
    ['Paul', 'Sledgehammer', 'Peter Gabriel', 'So', 'Funk', '1986']
```

To interact well with other functions, we will not attempt to test your code by calling `add_entry` with the same argument `person_name` *without removes in between*. It does not matter where in the list you add the new entry so long as you can successfully retrieve that entry with a later `find_entry` call using the same `person_name`.

2.3 remove_entry(database_list, person_name)

Given a string `person_name` in the same manner as with `find_entry` and `add_entry`, update `database_list` to remove the list containing the song data for the person named `person_name`. If there is no entry with a name matching `person_name`, do nothing. Like with `add_entry`, this function should **change the database `database_list` in place** but **not return anything**.

HINT: one approach could be to use another function you've written for this assignment (though be sure to check the output!).

2.4 replace_by_name(database_list, person_name, new_info)

Given a string `person_name` (in the same format as above), replace the song data for `person_name` in `database_list` with the new data contained within `new_info`.

There are *many* possible approaches to this, including the use of other functions you write for this homework. Whether you, for example, do a "full remove" on the existing entry and add a new one, or change an existing entry in place, what we'll look for is being able to see the new information we expect when we call `find_entry` on the same `person_name` after calling `replace_by_name`.

2.5 Helper functions

Each of these functions work with a *specific list* within the database that was previously described; that is, each of the following functions should work "directly" on a list of the following format or similar:

```
['Rob', 'On Top of the World', 'Imagine Dragons', 'Night Visions', '2013']
```

Each of these functions can be implemented in a very compact structure; in fact, it is probably preferable to do so.

2.5.1 get_favorite_song_name(song_info_list)

Given a list `song_info_list` of format described at the beginning of this subsection, return the **name of the song** (*not* the name of the person whose favorite song it is).

2.5.2 get_favorite_song_artist(song_info_list)

Given a list `song_info_list` of format described above, return the artist the song is by.

2.5.3 get_favorite_song_album(song_info_list)

Given a list `song_info_list` of format described above, return the album the song is from.

2.5.4 get_favorite_song_genre(song_info_list)

Given a list `song_info_list` of format described above, return the genre of the song.

2.5.5 get_favorite_song_year(song_info_list)

Given a list `song_info_list` of format described above, return the year the song was released in. *Note:* there is no need to return this as an `int`.

2.6 Reflection

In a separate PDF document (preferably using about 150-250 words total), answer the following two questions:

1. For each of the functions you implemented in this assignment, briefly describe ***A)*** whether the function can be used as-is using *tuples* instead of *lists*, OR ***B)*** what changes should be made to the function so that the function can be work ith tuples.

2. Do you feel this assignment was an effective way to apply what you learned in class? All feedback is appreciated, and specific feedback is encouraged.

Keep the following in mind as you write your code, as you will be graded on these aspects:

- Don't forget your header at the top of your `.py` file (see section 2.1)
- Cite your resources clearly or state you did not use any
- Use appropriate variables names throughout your program
- Include comments throughout your code to explain what you are doing (see example for an in-line comment below:)

```
foods = ['apple', 'pear']  # This is a list of foods
```

- Document all of your functions with **multi-line docstrings**. Note that the following is what we are referring to:

```
def func(first_param, second_param):
    '''
    First line - Description of the function - what does it do?
    :param first_param: what is it's data type,
        what does this parameter represent in the function?
    :param second_param: what is it's data type,
        what does this parameter represent in the function?
    :return: what is it's data type,
        what does the return value represent in the function?
    '''
    # BODY OF CODE HERE
```

(Remember, multi-line docstring comments are **not** a replacement for regular in-line comments. There is a place and need for both!)

3 Submission and Collaboration Policy

3.1 Your Submission–Comment Header

Your `.py` file should contain the following information (header) at the **top** of your file:

```
# NAME: e.g I. Lv. Sneks
# COMPUTING ID: e.g. ils3py@virginia.edu
# PA NUMBER and NAME: e.g. PA## - Name of the Assignment
# Resources used (if applicable):
```

For **Resources**, include URLs to any online resources where you studied or reviewed code that is specific to these problems, including any physical textbooks you referenced. Include any other resources you used here. *Note on resources:* Please feel free to refer to the lecture slides, demos, and in-class labs to complete this assignment, which can all be located on Canvas.

Absolutely no collaboration with any fellow students (who are not current CS 1112 course TAs) is allowed. Your work must represent individual effort. You must write your own code: not *just* type it, but also compose it yourself as your own original work.

You must cite any and every source you consult, other than those explicitly provided by the course itself. If you work with, obtain or receive help from another source (Internet website, textbook, TA, tutor, online video, etc.), nothing should be copied or retyped into the submitted solution. References must be documented in a comment in the code on the assignment. Any copied work is an Honor Code violation.

3.2 Gradescope

3.2.1 Submitting on Gradescope

1. Go to Gradescope (linked in Canvas course website)
2. Click on **PA05 – Song Manager** in the assignment list on Gradescope
3. Upload your Python program as `song_manager.py`.
4. Write your reflection in a separate document and save/download it as a **PDF** (please ask us if you are unsure how to save your file in PDF format. We are happy to help!)
5. Submit to the same Gradescope assignment PA05 – Song Manager
6. Ensure you have uploaded **two (2) files** before you submit!

You must match our file name, song_manager.py, ***exactly*** .

If you do not match our filename exactly, our autograder will be unlikely to find it, and it will therefore be unable to generate useful feedback to aid you on your submission.

Helpful guidance: Do not wait until the last minute to start this assignment! Remember you can **submit multiple times** to Gradescope. Look at the feedback you receive and then go back and fix your code, then you can resubmit. There is NO penalty for submitting multiple times to Gradescope. If you face any difficulties or have questions, post on Piazza. Also, don't hesitate to reach out to your professor or teaching assistants (TAs) for guidance. We are here to help!