Assignment 3

Due Dec 3 by 4pm **Points** 8 **Available** after Nov 2 at 9am

CSC108H Assignment 3

Deadline: Tuesday December 3 2019 by 4:00pm

Late policy: There are penalties for submitting the assignment after the due date. These penalties depend on how many hours late your submission is. Please see the syllabus on Quercus for more information.

Introduction

In this assignment, you will write a club recommendation system for a social network. Based on the scoring system described below, your program will recommend clubs that a person may wish to join.

This handout explains the problem you are to solve, and the tasks you need to complete for the assignment. Please read it carefully.

Goals of this Assignment

- Write function bodies using dictionaries and file reading.
- Write code to mutate lists and dictionaries.
- Use top down design to break a problem down into subtasks and implement helper functions to complete those tasks.
- Write tests to check whether a function is correct.

Files to Download

Please download the **Assignment 3 files** and extract the zip archive.

- Starter code:
  - `club_functions.py`

    This file contains the headers for the functions you will need to write for this assignment, and a few completed function docstrings. You will start development here, using the Function Design Recipe process to implement each required function. The functions in this file can call each other, and several of them will also be called by the main program (`club_finder.py`). You can, and should, write some helper functions in this file that are called by your required functions.

  - `test_get_average_club_count.py` and `test_get_last_to_first.py`

    We are providing the beginning of two unit test files, `test_get_average_club_count.py` and `test_get_last_to_first.py`. These files will contain your unit tests for the `get_average_club_count` and `get_last_to_first` functions, respectively, that are described below.

- Data: `profiles.txt`

  The `profiles.txt` file contains social network data. This is sample data, and you should not modify this file. You may want to create your own data files to supplement this file for testing. See the next section for instructions on how to interpret this data.

- Main Program: `club_finder.py`

  The file contains a program that loads some data and then calls some functions that you will implement in `club_functions.py`. **You do not need to modify this file.** After you have implemented all of your `club_functions.py` functions, you can run this program to interactively test your code using different profiles.

- Checker: `a3_checker.py`

  We have provided a checker program that you should use to check your code. See **below** for more information about `a3_checker.py`.

Data Format

The profile information for people in the social network is stored in a file. Your program will need to read the data from the file and process it.

The file contains 0 or more profiles. Each profile has the following format:

- 1 line containing the person's name
- then 0 or more lines containing the names of this person's clubs (one club per line)
- and finally 0 or more lines containing the names of this person's friends (one friend per line)

The profile file format adheres to the following rules:

- There is exactly one blank line between profiles.
- Club names do not contain commas.
- All lines that contain a person's name are in the following format:

  ```
  LastName, FirstName(s)
  ```

  You may assume that no person has a comma as part of their `LastName` or `FirstName(s)`. There is one comma in a line containing a person's name.

- Every person has a single (non-empty) last name (not a good assumption in general, but we'll make it for this assignment). A person may have a single (non-empty) first name or more than one word in their first name, separated by spaces.

  For example, the following are all valid lines that contain people's names in the profiles file:

  - `Mandela, Nelson`
  - `King-Noel, Augusta Ada`

- `Gandhi, Mohandas K.`

The starter zip file contains an example profile file named `profiles.txt`.

Your Tasks

You are required to:

1. write six required functions in `club_functions.py` and helper functions as appropriate (see marking scheme)
2. write `unittest`s for two of the required functions (`get_average_club_count` and `get_last_to_first`).

Data structures and ordering

Alphabetical Order

In the functions below, where alphabetical order is specified, you should use the `list` method `sort`. As a result, uppercase names like `'DJ Tanner'` would come before mixed-case names like `'Danny Tanner'`.

Data Structures

A "person to friends" dictionary (`Dict[str, List[str]]`) is our data structure for representing the mapping of people to their friends.

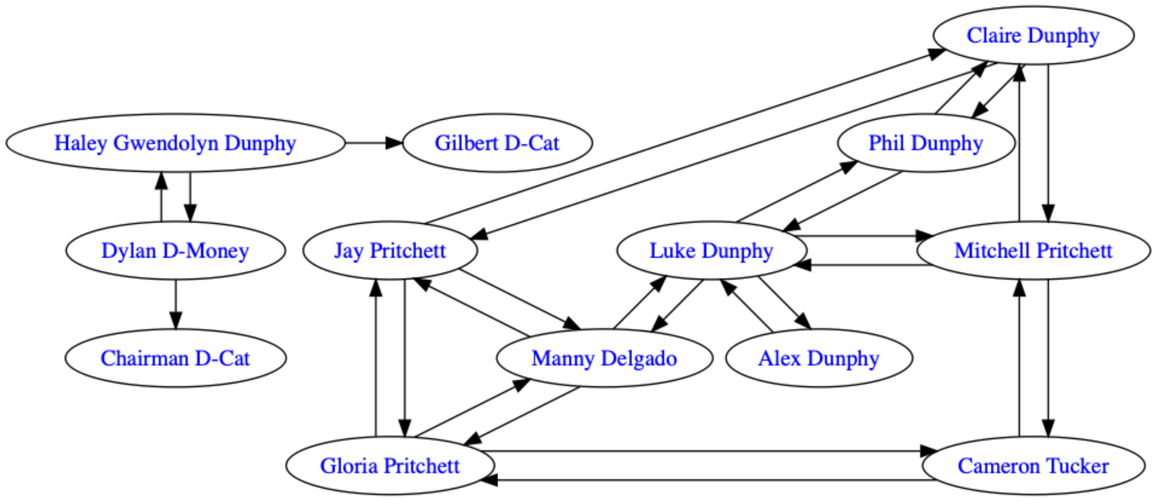
- **key:** person's name (`str`) in the format:

```
FirstName(s) LastName
```

- **value:** list of friends' names in `FirstName(s) LastName` format (`List[str]`), sorted in alphabetical order by `FirstName(s) LastName`. Use the `list` method `sort` to put the list of friends' names into alphabetical order.
- **example:** this dictionary represents the "person to friends" data from `profiles.txt`:

```
       {'Jay Pritchett': ['Claire Dunphy', 'Gloria Pritchett', 'Manny Delgado'],
        'Claire Dunphy': ['Jay Pritchett', 'Mitchell Pritchett', 'Phil Dunphy'],
        'Manny Delgado': ['Gloria Pritchett', 'Jay Pritchett', 'Luke Dunphy'],
        'Mitchell Pritchett': ['Cameron Tucker', 'Claire Dunphy', 'Luke Dunphy'],
        'Alex Dunphy': ['Luke Dunphy'],
        'Cameron Tucker': ['Gloria Pritchett', 'Mitchell Pritchett'],
        'Haley Gwendolyn Dunphy': ['Dylan D-Money', 'Gilbert D-Cat'],
        'Phil Dunphy': ['Claire Dunphy', 'Luke Dunphy'],
        'Dylan D-Money': ['Chairman D-Cat', 'Haley Gwendolyn Dunphy'],
        'Gloria Pritchett': ['Cameron Tucker', 'Jay Pritchett', 'Manny Delgado'],
        'Luke Dunphy': ['Alex Dunphy', 'Manny Delgado', 'Mitchell Pritchett', 'Phil Dunphy']}
```

Visually, we can think of this person to friends dictionary as the following network, where an arrow from Jay Pritchett to Gloria Pritchett indicates that Jay Pritchett has a friend named Gloria Pritchett.



A "person to clubs" dictionary (`Dict[str, List[str]]`) is our data structure for representing the mapping of people to the clubs they are members of.

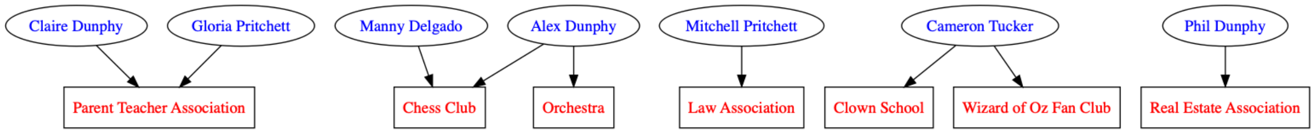
- **key:** person's name (`str`) in the format:

```
FirstName(s) LastName
```

- **value:** list of clubs that person is a member of (`List[str]`), sorted in alphabetical order. Use the `list` method `sort` to put the club names into alphabetical order.
- **example:** this dictionary represents the "person to clubs" data from `profiles.txt`

```
    {'Claire Dunphy': ['Parent Teacher Association'],
     'Manny Delgado': ['Chess Club'],
     'Mitchell Pritchett': ['Law Association'],
     'Alex Dunphy': ['Chess Club', 'Orchestra'],
     'Cameron Tucker': ['Clown School', 'Wizard of Oz Fan Club'],
     'Phil Dunphy': ['Real Estate Association'],
     'Gloria Pritchett': ['Parent Teacher Association']}
```

Visually, we can think of this person to clubs dictionary as the following network, where an arrow from Manny Delgado to Chess Club indicates that Manny Delgado is a member of the Chess Club.



Required Testing (`unittest`)

Write (and submit) unittest test files for functions `get_average_club_count` and `get_last_to_first`. These tests should be implemented in the appropriately named starter files.

We will evaluate the completeness of your test files by running them against flawed implementations we have written to see how many errors you catch. Avoid redundant tests. The goal is to catch all of our errors without extra, unnecessary tests.

Your unittest testfiles should stand alone: they should require no additional files (like a profile file). Instead, you should define appropriate test values in the file, such as a dictionary that might be generated from reading a profile file, to use for your tests. You may assume that the folder that contains your unittest test files also contains a `club_functions.py` file.

Recall that floating-point numbers are approximations of real numbers. To compare `float`s, you should use the `assertAlmostEqual` method, as we have done in the starter code.

Required Functions

This section contains a table with detailed descriptions of the 6 functions that you must complete. You'll need to add a second example to the docstrings for each function in the starter code.

We provided one helper function in the starter code that you may use. You should follow the approach we've been using on large problems recently and write additional helper functions to break these high-level tasks down. Each helper function must have a clear purpose. Each helper function must have a complete docstring produced by following the Function Design Recipe. You should test your helper functions to make sure they work.

Functions to write for A3

Function name: (Parameter types) -> Return type	**Full Description (paraphrase to get a proper docstring description)**
`load_profiles:` `(TextIO)` -> `Tuple[Dict[str, List[str]], Dict[str, List[str]]]`	The parameter refers to a file that is open for reading. The data in the file is in the Data Format described above. This function must build a "person to friends" dictionary and a "person to clubs" dictionary with the data from the open file, and return a two-item tuple containing the dictionaries (the first item is the "person to friends" dictionary and the second item is the "person to clubs" dictionary). **Note: it may turn out that person A has person B in their friends list, while person B does not have person A in their friends list.** Notes: • You do not need to finish this function before working on the rest of the assignment. If you get stuck on this function, move on to the other functions and come back to it. • If a person does not have any friends, then they must not appear in the "person to friends" dictionary. That is, if a person is only listed as a friend and does not have friends of their own, they should not appear as a key in the dictionary.

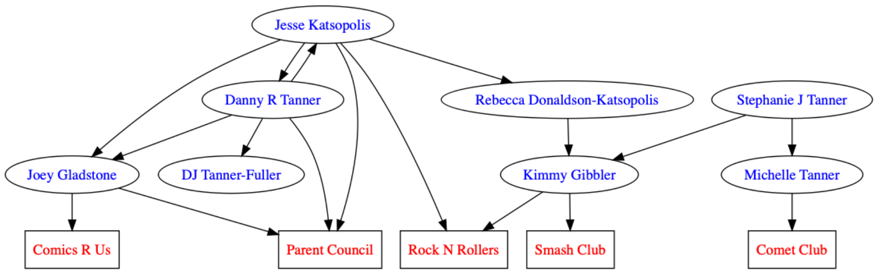
	• Similarly, if a person is not a member of any clubs, then they must not appear in the "person to clubs" dictionary.
`get_average_club_count: (Dict[str, List[str]]) -> float`	The parameter represents a "person to clubs" dictionary. This function must return the average number of clubs that people in this dictionary are members of.
`get_last_to_first: (Dict[str, List[str]]) -> Dict[str, List[str]]`	The parameter represents a "person to friends" dictionary. This function must return a "last name to first names" dictionary built from the data in the given dictionary. The keys in the returned dictionary should be the last names of people in the given dictionary. (Remember: every person has exactly one last name.) The values are lists of the first names (sorted in alphabetical order) of people with a given last name. The returned dictionary should contain every person whose name appears in the given dictionary: the person's last name as one of the dictionary keys, and the person's first name as an element in one of the dictionary values. Names in the list should be unique: no one should be listed more than once. Use the `list` method `sort` to put the lists of first names into alphabetical order.
`invert_and_sort: (Dict[object, object]) -> Dict[object, list]`	The parameter represents a dictionary, in which either the values are all lists or none of the values are lists. This function must return a new dictionary that is the given dictionary inverted: each key is a value or an item from a values list from the given dictionary, and each value is a list of the corresponding keys from the given dictionary. The items in the new dictionary's values lists are sorted. For example, given the following argument (based on `profiles.txt`):

```
{'Claire Dunphy': ['Parent Teacher Association'],
 'Manny Delgado': ['Chess Club'],
 'Mitchell Pritchett': ['Law Association'],
 'Alex Dunphy': ['Chess Club', 'Orchestra'],
 'Cameron Tucker': ['Clown School', 'Wizard of Oz Fan Club'],
 'Phil Dunphy': ['Real Estate Association'],
 'Gloria Pritchett': ['Parent Teacher Association']}
```

the function should return:

```
{'Parent Teacher Association': ['Claire Dunphy', 'Gloria Pritchett'],
 'Chess Club': ['Alex Dunphy', 'Manny Delgado'],
 'Law Association': ['Mitchell Pritchett'],
 'Orchestra': ['Alex Dunphy'],
 'Clown School': ['Cameron Tucker'],
 'Wizard of Oz Fan Club': ['Cameron Tucker'],
 'Real Estate Association': ['Phil Dunphy']}
```

`get_clubs_of_friends: (Dict[str, List[str]], Dict[str, List[str]], str) -> List[str]`	The first parameter represents a "person to friends" dictionary, the second parameter represents a "person to clubs" dictionary, and the third parameter represents the name of a person (in the same format as the dictionary keys). The function should return a list of clubs to which the given person's friends are members of. The returned list should be sorted in alphabetical order. The list should not contain clubs that the given person is a member of. The clubs should appear in the returned list once for each of the person's friends that are members of it. (That is, clubs may be included more than once in the returned list.) Use the `list` method `sort` to put the clubs into alphabetical order.
`recommend_clubs: (Dict[str, List[str]], Dict[str, List[str]], str) -> List[Tuple[str, int]]`	The first parameter represents a "person to friends" dictionary, the second parameter represents a "person to clubs" dictionary, and the third parameter represents a person's name (in the same format as the dictionary keys). Using the recommendation system described **below**, the function should return the club recommendations for the given person as a list of 2-element tuples. The first element of each tuple is a potential club's name (in the same format as the dictionary keys) and the second element is that potential club's score. Only potential clubs with non-zero scores should be included in the list. The recommendations should be sorted from highest to lowest score. If multiple clubs have the same score, they should be sorted alphabetically (by the clubs' names). The person given as the first argument may be a key in zero, one, or both of the given dictionaries. **Recommendation score** Imagine that a person wants to get recommendations for *potential clubs* — clubs they are not a member of but may wish to join. The social network calculates a *recommendation score* for each potential club. For a particular person, each potential club starts out with a score of 0 and is scored using the following point system: • Add 1 point to the potential club's score for each of the person's friends who are members of the potential club. • Add 1 point for every member of the potential club, who is in at least one different club with the person. Examples (refer to the diagram below, which represents the data in P2F and P2C in the starter code):



When making recommendations for Jesse Katsopolis, the result is: [('Comics R Us', 2), ('Smash Club', 1)]

- Comics R Us gets one point because Jesse Katsopolis's friend Joey Gladstone is a member of Comics R Us.
- Comics R Us gets a second point because Joey is a member of Comics R Us and both Jesse and Joey are members of Parent Council.
- Smash Club gets one point because Kimmy is a member of Smash Club and Kimmy is in a different club (Rock N Rollers) with Jesse.

When making recommendations for Stephanie J Tanner, the result is: [('Comet Club', 1), ('Rock N Rollers', 1), ('Smash Club', 1)]

- Comet Club gets one point because Stephanie's friend Michelle Tanner is a member of Comet Club.
- Smash Club gets one point because Stephanie's friend Kimmy Gibbler is a member of Smash Club.
- Rock N Rollers gets one point because Stephanie's friend Kimmy Gibbler is a member of Rock N Rollers.

Hint: There are two ways a potential club's score can increase. Consider creating a helper function for each way.

A3 Checker

We are providing a checker module (`a3_checker.py`) that tests two things:

- whether your code follows the **Python Style Guidelines**, and
- whether your functions are named correctly, have the correct number of parameters, and return the correct types.

To run the checker, open `a3_checker.py` and run it. Note: the checker file should be in the **same** directory as your `club_functions.py`, as provided in the starter code zip file. Be sure to scroll up to the top and read all messages.

If the checker passes for both style and types:

- Your code follows the style guidelines.

- Your function names, number of parameters, and return types match the assignment specification. **This does not mean that your code works correctly in all situations.** We will run a *different* set of tests on your code once you hand it in, so be sure to thoroughly test your code yourself before submitting.

If the checker fails, carefully read the message provided:

- It may have failed because your code did not follow the style guidelines. Review the error description(s) and fix the code style. Please see the **PyTA documentation (http://www.cs.toronto.edu/~david/pyta/)** for more information about errors.
- It may have failed because:
  - you are missing one or more function,
  - one or more of your functions is misnamed,
  - one or more of your functions has the incorrect number or type of parameters, or
  - one of more of your function return types does not match the assignment specification.

  Read the error message to identify the problematic function, review the function specification in the handout, and fix your code.

Make sure the checker passes before submitting.

Running the checker program on Markus

In addition to running the checker program on your own computer, run the checker on MarkUs as well. You will be able to run the checker program on MarkUs once every 12 hours (note: we may have to revert to every 24 hours if MarkUs has any issues handling every 12 hours). This can help to identify issues such as uploading the incorrect file.

First, submit your work on MarkUs. Next, click on the "Automated Testing" tab and then click on "Run Tests". Wait for a minute or so, then refresh the webpage. Once the tests have finished running, you'll see results for the Style Checker and Type Checker components of the checker program (see both the Automated Testing tab and results files under the Submissions tab). Note that these are not actually marks -- just the checker results. If there are errors, edit your code, run the checker program again on your own machine to check that the problems are resolved, resubmit your assignment on MarkUs, and (if time permits) after the 24 hour period has elapsed, rerun the checker on MarkUs.

Testing your Code

It is strongly recommended that you test each function *as soon as* you write it. As usual, follow the Function Design Recipe (we've provided the function name and types for you) to implement your code. Once you've implemented a function, run it against the examples in your docstrings and the unit tests you've defined.

Additional requirements

- Do **not** call `print`, `input`, or `open`, except within the `if __name__ == '__main__'` block.
- Do **not** use any `break` or `continue` statements.
- Do **not** modify or add to the import statements provided in the starter code.

- Do **not** add any code outside of a function definition.
- Do **not** mutate objects unless specified.
- Do **not** use Python language features for sorting that we haven't covered in this course, like the optional parameter `key` or the function `sorted`.

Marking

These are the aspects of your work that will be marked for Assignment 3:

- **Correctness (70%):** Your functions should perform as specified. Correctness, as measured by our tests, will count for the largest single portion of your marks. Once your assignment is submitted, we will run additional tests, not provided in the checker. Passing the checker **does not** mean that your code will earn full marks for correctness.
- **Testing (10%):** We will run the `unittests` that you submit on a series of flawed (incorrect) implementations we have written. Your testing mark will depend on how many of the flawed implementations your unittests catch, whether they successfully pass a working (correct) implementation, and whether your test files contain redundant (unnecessary) tests.
- **Coding style (20%):**
  - Make sure that you follow the **Python style guidelines** that we have introduced and the Python coding conventions that we have been using throughout the semester. Although we don't provide an exhaustive list of style rules, the checker tests for style are complete, so if your code passes the checker, then it will earn full marks for coding style with two exceptions: docstrings and use of helper functions may be evaluated separately. For each occurrence of a **PyTA error (http://www.cs.toronto.edu/~david/pyta/)** , a 1 mark (out of 20) deduction will be applied. For example, if a C0301 (line-too-long) error occurs 3 times, then 3 marks will be deducted.
  - Your program should be broken down into functions, both to avoid repetitive code and to make the program easier to read. If a function body is more than about 20 statements long, introduce helper functions to do some of the work -- even if they will only be called once.
  - All functions, including helper functions, should have complete docstrings including preconditions when you think they are necessary.
  - Also, your variable names and names of your helper functions should be meaningful. Your code should be as simple and clear as possible.

No Remark Requests

No remark requests will be accepted. A syntax error could result in a grade of 0 on the assignment. Before the deadline, you are responsible for running your code and the checker program to identify and resolve any errors that will prevent our tests from running.

What to Hand In

The very last thing you do before submitting should be to run the checker program one last time.

Otherwise, you could make a small error in your final changes before submitting that causes your code to receive zero for correctness.

Submit `club_functions.py`, `test_get_average_club_count.py`, and `test_get_last_to_first.py` on MarkUs. Remember that spelling of filenames, including case, counts: your file must be named exactly as above.