15-110 Fall 2019
Hw 07

Out: Tuesday 5th November, 2019 at 14:00 AST
Due: Wednesday 13th November, 2019 at 23:55 AST

Introduction

In this homework you will practice with dictionary and sets concepts and little with files.

The total number of points available from the questions is 115, 15 points are *bonus* points (i.e., you only need 100 points to get the maximum grade).

In your `.zip` file (see general instructions below), you need to include only the file `hw07.py` with the python functions answering the questions. In the handout you have found a file `hw07.py` with the functions already defined but with an empty body (or partially filled). You have to complete the body of each function with the code required to answer to the questions.

General Instructions for Submitting the Assignments

Submissions are handled through Autolab, at https://autolab.andrew.cmu.edu/courses/15110q-f19

You are advised to create on your computer/account a folder named `110-hw`. For each new homework, you should create a new sub-folder named `01`, `02`, etc. where you can put the files related to the homework. In this way you work will be nicely organized and information and files will be easily accessible.
You can also create an equivalent struture for the *laboratories*, where in this case the root folder should be named `110-lab`.

When you are ready with the homework and want to submit your solutions, you need to go in the current homework folder (e.g., `01`), *select all files you will submit* (that can include both `.pdf` files with written answers to questions and python code files, `.py`) and compress them in one single `.zip` file.[1]

According to the OS you are using, you migh have different options for making the zip file. For instance, on Windows, after selection of the files, you should right-click and select `Send to: Compressed folder`, while on macOS, you can select `Compress` on the menu appearing from the right-click.

The compression action will produce a zip file containing the files to be handed in for the assignment. The file should be named `hwXX-handin.zip` (e.g., for this homework, the name of the file should be `hw07-handin.zip`). Then, open Autolab, find the page for this assignment, and submit your `hw07-handin.zip` file via the "Submit" link.

☛ The number of submissions is limited to 5. The last submission is the one that will be graded.

Style

Part of your grade on assignments are style points, that can be lost if your code is too disorganized, unreadable or unnecessarily complicated. To avoid loosing style points, please follow the guidelines at https://web2.qatar.cmu.edu/cs/15110/resources/style.pdf.

[1] The (single) zip file is needed, even when the files handed for the assignment consists of one individual file.

1 Single Eaters

As you probably already know, some animals only eat meat (carnivores), some animals only eat plants (herbivores), and some animals eat both (omnivores).

Problem 1.1: (20 points)

Implement the function `single_eaters(eatsMeat, eatsPlants, animalList)`, which given a list of animals that eat meat (`eatsMeat`), a list of animals that eat plants (`eatsPlants`), and a list of animals (`animalList`) returns the **set** of all animals from `animalList` that eat either meat or plants but not both.

So, for example: `single_eaters(['vulture', 'lion', 'bear'], ['bear','gazelle'],['bear', 'gazelle', 'vulture', 'lion', 'camel'])` returns {`'vulture', 'lion', 'gazelle'`}.

Note that even though this problem is written with regards to lists of animals, your solution should work with lists of anything. This means that it could also function with lists of numbers: `single_eaters([1, 15, 20], [20, 3], [1, 15, 20, 3, 102])` returns {`1, 3, 15.`}

2 Friends

Consider a dictionary that stores a list of people that consider each other friends:

```
d = dict()
d["fred"] = set(["betty", "barney"])
d["wilma"] = set(["fred", "betty"])
d["betty"] = set(["barney"])
d["barney"] = set()
```

In this example, fred considers his friends to be betty and barney; wilma considers her friends to be fred and betty; betty considers her only friend to be wilma, and barney believes he has no friends.

We will consider someone's Bad Friend Score to be the number of people who they consider friends but whom do not consider them a friend. For example, wilma's bad friend score is one because she considers both fred and betty to be her friends, but fred does not consider her a friend back.

For the above dictionary:

```
bad_friend_score(d, "fred") == 2
bad_friend_score(d, "wilma") == 2
bad_friend_score(d, "betty") == 1
bad_friend_score(d, "barney") == 0
```

Problem 2.1: (20 points)

Implement the function `bad_friend_score(d, person)` that takes a dictionary of the above form and a person to check and returns that person's bad friend score.

We will consider person A's antisocial score to be the number of people B such that A does not list B as a friend, but B does list A.

For the dictionary given above:

Person	Antisocial Score	Reason
wilma	0	Nobody likes wilma, so wilma cannot dislike anyone who likes her
fred	1	wilma likes fred, but fred does not like wilma
betty	2	fred and wilma like betty, but betty does not like either of them
barney	2	fred and betty like barney, but barney does not like either of them

Your goal is to write the function `antisocial_score(d,person)` that takes a dictionary of the given form, and a person who you may assume is in the dictionary, and returns the antisocial score of that person.

In order to help you solve the larger problem, you should first write a helper function.

Problem 2.2: (15 points)

Write the helper function `likes(d, person)` that takes a dictionary of the given form, and a person you may assume is in the dictionary, and returns the set of people who list that person as a friend. Check the skeleton file test case for examples.

Now that you have written your helper function, you can move on to solving the main problem.

Problem 2.3: (15 points)

Use your helper function to help you write the function `antisocial_score(d,person)` that takes a dictionary of the given form, and a person who you may assume is in the dictionary, and returns the antisocial score of that person.

3 Most Common Word

Processing large amounts of text can be a very useful exercise. Imagine that you have a large amount of text and you want to figure out what the most frequently occuring word is in that text. This might be a somewhat complicated task, because normal text has punctuation, the same word may be written with different upper and lowercase letters, etc. For example, "monkey", "Monkey", "monkey,", and "Monkey." are all the same word despite varying letter case and punctuation.

Problem 3.1: (30 points)

Implement the function `most_common_word(text)`, which, given a string of text returns the word that most frequently occurs (without respect to upper and lowercase) in the form it most frequently occurs in (with respect to upper and lowercase).

For example, consider the text `"hi␣bob␣there␣Hi␣bob␣hI␣Hi␣bob"`. The most frequently occurring word in the text (without respect to case) is `"hi"`, which occurs 4 times. The form it occurs most frequently in is `"Hi"`, which occurs twice. So, that means that: `most_common_word("hi␣bob␣there␣Hi␣bob␣hI␣Hi␣bob")` should return `"Hi"`.

Further complicating things is the fact that the text may contain punctuation, numbers, special characters, and extra space characters, and those should not affect the answer. (You should ignore anything that is not a letter.) So, that means that: `most_common_word("hi,␣bob␣$there␣{Hi␣bob␣hI.␣␣␣Hi%␣bob")` should still return `"Hi"`.

You must not try to write all of your code in one function. You should alwyas write helper functions that each do small parts of the task and then use those to solve the problem. Here is the set of helper functions that you must implement:

- `filter_text(text)`: Given a **string** of `text`, return a new string that contains only the letters and whitespace from `text`. (This filters our all of the characters we don't care about.)
- `most_frequent_ignore_case(words)`: Given a **list** of `words`, return the word that occurs most frequently in the list, ignoring case. To keep track of the frequency of each word as the words are processed, you should use a dictionary where the key is a word and the value is its frequency.
- `most_frequent_form(words, word)`: Given a **list** of `words` and a `word` in lowercase, return the most frequent form of `word` in `words`. You can use the same idea as before: a dictionary to keep track of word frequencies. Except this time capitalization will matter, and you only need to count the words that are the same as **word**.

Remember: Build your solution incrementally instead of trying to solve the entire problem at once. First, build a solution that can find the most common word (without regards to case) from a simple text with no special characters. Once that works, then improve it to also work in the case of strange punctuation. Once that works, improve it again to return the most common form of the word.

The following functions may be useful for the tasks above[2]

- `s.split(sep)`: Return a list of the words in the string, using `sep` as the delimiter string. If `sep` is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace.
- `s.lower()`: Return a copy of the string with all the cased characters converted to lowercase.
- `s.upper()`: Return a copy of the string with all the cased characters converted to uppercase.
- `s.isalpha()`: Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.
- `s.isspace()`: Return true if there are only whitespace characters in the string and there is at least one character, false otherwise.

[2]Do not feel like you are supposed to use all of them, though!

4 Grades

Now that you know how files work, suppose we are keeping track of students' grades in a file that looks like this:

```
rick  , 10, 10 , 10 , 9.7, 8.7
morty , 8 , 7.5, 10 , 9  , 7.6
beth  , 7 , 9.6, 8.5, 10
jerry , 6 , 5.4, 3.8, 10
summer, 10, 9.5, 8.5, 5  , 7.2, 8
```

Each student is on a separate line, where the first element is their name, followed by their scores, separated by spaces. Notice that not all students have the same number of assignments.

Problem 4.1: (15 points)

Implement the function `compute_grade(filename)` that takes as input the name of the file that contains the grades, and writes a file named `grades.txt` where each line corresponds to a student, formatted as follows:

- the name of the student as a string of 15 characters;
- the average grade of the student as a floating point number of 5 digits, where 2 correspond to the decimal places;
- the letter grade A, B, C, D, or R, depending on the average x of the student:
  - A if $x \geq 9$
  - B if $9 < x \geq 8$
  - C if $8 < x \geq 7$
  - D if $7 < x \geq 6$
  - R otherwise

The function should return the string that was written in the file.