CSCI 1100 — Computer Science 1 Homework 4
Loops and Lists

Overview

This homework is worth **100 points** total toward your overall homework grade. It is due Thursday, February 28, 2018 at 11:59:59 pm. As usual, there will be a mix of autograded points, instructor test case points, and TA graded points. There are two parts in the homework, each to be submitted separately. All parts should be submitted by the deadline or your program will be considered late.

See the handout for Homework 3 for a discussion on grading and on what is considered excessive collaboration. These rules will be in force for the rest of the semester.

You will need the utilities and data files we provide in `hw04_files.zip`, so be sure to download this file from the `Course Materials` section of Submitty and unzip it into your directory for HW 4. Module `hw4_util.py` is written to help you read information from the files. You do not need to know how functions provided in `hw4_util.py` are implemented (but feel free to examine the code, if you are interested), you can simply use them.

Final note, you will need to use loops in this assignment. We will leave the choice of the loop type up to you. Please feel free to use `while` loops or `for` loops depending on the task and your personal preference.

Part 1: Is this Crazy Password Valid? (40 pts)

A good password (in this strange system) should have the following properties:

Rule 1: The password should be between 4 and 25 characters inclusive, starting with a letter.

Rule 2. The password should not be contained in a list (selected by the user) that contains passwords commonly used by many people thus making these passwords easy to guess. Comparisons with passwords from the list of common passwords should be case insensitive. For example, if the password entered by the user is `P@sSw0rD` and there is a common password `p@ssw0rd` in the list, Rule 2 is not satisfied.

Rule 3. The password should have at least one of the following characters `@ $` and no `%`.

Rule 4: The password should have at least one upper and one lower case character, or it should contain at least one of 1, 2, 3 or 4.

If it contains upper and lower case characters, we don't care about numbers 1, 2, 3, 4. If it contains 1, 2, 3 or 4, we don't care about upper or lower case characters.

Rule 5: Each upper case character (if there is any) must be immediately followed by at least one underscore symbol (`_`).

Rule 6: Each numerical digit, if there is any, must be less than 5. (So, `pa$$35` is not valid because of 5.) Note that in Rule 4 you checked the existence of at least one number in the 1-4 range, now you want to check that all numbers are in this range.

Write a program that validates a password entered by the user against this set of rules. The program should first determine the number of available lists with common passwords (as described below), prompt the user to enter a correct number to select the list, and then ask for a password. If the password is a valid one (satisfying all the above rules), print a corresponding message to the user. If the password is not valid but satisfies Rule 1, suggest a valid password by taking the first 3 and the last 3 characters and appending `42` between them (note the password constructed this

way may not be valid with respect to the above rules, but randomness is always good for security). Otherwise, report the password as not valid.

In order to check the password against Rule 2, you will first need to determine how many lists with common passwords are available to you. Use function `get_number_of_password_lists()` provided in `hw4_util` module. Make sure you have your code for this part of the assignment, the `hw4_util.py` file, and two data files, `password_list_top_25.txt` and `password_list_top_100.txt`, all residing in the same directory. If everything is set up correctly, the following code:

```
import hw4_util
lists_count = hw4_util.get_number_of_password_lists()
print(lists_count)
```

should print 2, indicating that you have 2 lists (numbered 0 and 1) with common passwords available to you. You can obtain any of these lists by using `get_password_list()` function from the `hw4_util` module and specifying the list number, as in the following example (we slice the list to only show the first 5 items for brevity):

```
import hw4_util
passwords = hw4_util.get_password_list(0)
print(passwords[0:5])
```

which outputs `['123456', 'Password', '12345678', 'qwerty', '12345']`.

Note that when you submit your code on Submitty we will be using lists of common passwords which are different from the ones we provide to you, and we will be using more than two lists. Your code should not hardcode the number of lists available or the contents of the lists. Instead, use functions `get_number_of_password_lists()` and `get_password_list()` from `hw4_util`.

Hint: A very good way to solve this problem is to use booleans to store whether each rule is satisfied or not. Start slowly, write and test each part separately. Rules 1, 3, and 4 do not require looping at all. For Rule 2, you will need to loop through all items in the list of common passwords. For Rule 5, the simplest solution involves looping through the string once. The same is true for Rule 6. To construct the suggested password, you only need one line of code by using slicing.

There are some really useful string functions you can use: `str.islower()`, `str.isupper()` and of course `str.count()` and `str.find()`. Try `help(str)` to learn more about them.

Here is an example solution that involves using booleans to store the result and building on them:

```
is_long_enough = len(word) <= 25 and len(word) >= 4
is_lowercase = word.islower()
if is_long_enough and is_lowercase:
    print ("It is long enough and lower case")
```

Four examples of the program run (how it will look when you run it using Wing IDE 101) are provided in files `hw4_part1_output_01.txt`, `hw4_part1_output_02.txt`, `hw4_part1_output_03.txt`, and `hw4_part1_output_04.txt`. For all these examples, it is assumed that both lists of common passwords that we provide are available.

Test your code well and when you are sure that it works, please submit it on Submitty as a file named **hw4_part1.py** for Part 1 of the homework. You must use this filename, or your submission will not work on Submitty. You do **not** have to submit any of the files we have provided.

Part 2: ZIP Code Look up and Distance Calculation (60 pts)

As part of the Open Data Policy, the U.S. Government released a large number of datasets. Among these is a dataset that provides geographic coordinates for U.S. 5 digit ZIP Codes which we will be using for this assignment. For each ZIP Code, this dataset lists geographic coordinates (latitude and longitude), city, state, and county. Our goal is to use available data to develop a new data product that would allow users not only to query and search the existing data source but also to access additional functionality such as computing the distance between two locations.

Write a program that would allow users to lookup locations by ZIP Code, lookup ZIP Codes by city and state, and determine the distance between two locations designated by their ZIP Codes. The program interacts with the user by printing a prompt and allowing them to enter commands until they enter `'end'` at which point the program prints `Done` and finishes. If an invalid command is entered, the program prints `Invalid command, ignoring` and is ready to take the next command. All commands are case insensitive (i.e., can be typed using any case or a mixture of lower and upper case).

The following commands should be recognized:

1. **loc** allows the user to enter a ZIP Code, then looks up city, state, county, and geographic coordinates that correspond to the ZIP Code and prints this data. If the ZIP Code is invalid or not found in the dataset, the program prints an error message instead. Look at the following sample output:

```
Command ('loc', 'zip', 'dist', 'end') => loc
loc
Enter a ZIP Code to lookup => 19465
19465
ZIP Code 19465 is in Pottstown, PA, Chester county,
coordinates: (042°40'25.32"N,073°36'31.65"W)
```

   Note that coordinates are enclosed in parentheses and are separated by a comma; each coordinate is printed in integer degrees (three digits), followed by the ° symbol, integer minutes (two digits), followed by the ' character, integer and fractional seconds (two integer and two decimal digits), followed by the " character, and a cardinal direction letter (N, S, W, or E) with no spaces anywhere in the coordinate representation.

2. **zip** allows the user to enter city and state, then looks up the ZIP Code or codes (some cities have more than one) which correspond to this location and prints them; if city and/or state are invalid or not found in the dataset, it prints an error message instead.

```
Command ('loc', 'zip', 'dist', 'end') => zip
zip
Enter a city name to lookup => troY
troY
Enter the state name to lookup => ny
ny
The following ZIP Code(s) found for Troy, NY: 12179, 12180, 12181, 12182, 12183
```

3. **dist** allows the user to enter two ZIP Codes and computes the geodesic distance between the location coordinates; if any of the ZIP Codes entered is invalid or not found in the dataset, it prints a corresponding error message instead.

```
Command ('loc', 'zip', 'dist', 'end') => dist
dist
Enter the first ZIP Code => 19465
19465
Enter the second ZIP Code => 12180
12180
The distance between 19465 and 12180 is 201.88 miles
```

4. **end** stops fetching new commands from the user and ends the program.

Utility module `hw4_util` will give you some help. It provides a function `read_zip_all()` that returns a list each element of which is, in turn, a list that contains data about one ZIP Code. Try the following:

```
import hw4_util
zip_codes = hw4_util.read_zip_all()
print(zip_codes[0])
print(zip_codes[4108])
```

and it should output:

```
['00501', 40.922326, -72.637078, 'Holtsville', 'NY', 'Suffolk']
['12180', 42.673701, -73.608792, 'Troy', 'NY', 'Rensselaer']
```

Note that the data on a particular ZIP Code has the following fields: ZIP Code (string), latitude (float degrees), longitude (float degrees), city (string), state (string), and county (string).

Implementation Details

When grading your code, we will be importing it as a module and then verifying that it works correctly by calling two functions with specific names and comparing the return values with expected data. You will need to define these two functions by strictly following specifications outlined below:

1. `zip_by_location(zip_codes, location)` which finds the ZIP Code for a given location.
   Parameters:
   `zip_codes` a list of ZIP Codes data in the format, returned by `read_zip_all()`;

   `location` a two-element tuple where the first element is the city name and the second element is the state abbreviation, e.g., `('trOy', 'nY')`. Both elements are string values. City names and state abbreviations can be typed using any case or a mixture of lower and upper case.

   Return value:
   A list which contains the ZIP Code or codes for the specified location. Each ZIP Code is a string value. If the location is invalid, an empty list is returned.
   E.g., `['12179', '12180', '12181', '12182', '12183']`.

2. `location_by_zip(zip_codes, code)` which finds location information corresponding to the specified ZIP Code.

Parameters:

`zip_codes` a list of ZIP Codes and associated data in the format, returned by `read_zip_all()`;

`code` ZIP Code as a string value, e.g. `'12180'`.

Return value:
A five-element tuple `(latitude, longitude, city, state, county)`. Latitude and longitude are in fractional degrees (floating point values). All other elements are string values. If the ZIP Code is invalid, an empty tuple is returned.
E.g., `(42.673701, -73.608792, 'Troy', 'NY', 'Rensselaer')`.

It is required that you implement functions according to specifications given above. Remember, if your functions do not follow these requirements (e.g., you named them something else, or they expect an argument or return a value which are different from what is described above), our attempt to call your function might fail, and you will lose points.

You are also expected to define other functions, as necessary. You need to determine which functions to define and to design their specifications yourself, following the example we gave above for the two required functions. Do not forget to include function specifications as a docstring immediately below the function's definition in your code. Avoid copying and pasting repeated pieces of code. Those should be factored out as functions. You may lose points for excessive duplication of code.

Make sure that when your code is imported as a module, it does not print anything and does not attempt to ask the user to input any values. The main body of your program should be behind a corresponding `if` statement, so that the main code is executed only when your Python file is run directly and is not executed when your module is imported somewhere else.

Hints and Suggestions

Formatting your output is a big part of this program. Here are a few hints.

1. The ° symbol can be generated using another escape character similar to `\n` or `\"`. To generate the °, use `\xb0`. Just like the newline or a double quote, this is a single character.

2. You may find `tuple()` function useful if you need to convert a list to a tuple, e.g., `tuple(["RPI", "Private", 7962])`.

3. If you want to print leading `0`s before an integer, use the `{:0Nd}` where `N` is the total number of character positions you want the integer to occupy. So, a format of `:07d` will use 7 character positions to print your integer, padding the front of the integer with `0`s. I.e.,

   ```
   print("{:07d}".format(7))
   ```

   will give

   ```
   0000007
   ```

4. You will need to manage the latitude and longitude to convert among representations. In particular, you need to convert the fractional degrees in the ZIP Code list to degrees, minutes,

and seconds. There are `60` minutes in a degree and `60` seconds in a minute. When you convert, all of your latitudes and longitudes should be positive. Instead, use east (`E`) and west (`W`) designators of longitude; and north (`N`) and south (`S`) designators for latitude. Negative longitudes are west, positive are east. Negative latitudes are south, positives are north. In both cases, a latitude or longitude of `0` (the equator or the prime meridian, respectively) do not have a designator.

5. The distance between two points on the surface of Earth uses a simplified haversine formula for arc length on a sphere. You should implement Eq. 1 which assumes the Earth to be spherical. This formula gives the shortest surface distance d "as-the-crow-flies", ignoring Earth's relief.

$$\begin{aligned} \Delta latitude &= latitude_2 - latitude_1 \\ \Delta longitude &= longitude_2 - longitude_1 \\ a &= sin^2\left(\frac{\Delta latitude}{2}\right) + cos(latitude_1) \cdot cos(latitude_2) \cdot sin^2\left(\frac{\Delta longitude}{2}\right) \\ d &= 2R \cdot arcsin\left(\sqrt{a}\right) \end{aligned} \tag{1}$$

   Where $R = 3959.191$ miles is the radius of the Earth and all angle measurements are in **radians** (the ZIP Code data gives latitude and longitude in degrees). 360 degrees is equivalent to 2π radians.

An example of the program run (how it will look when you run it using Wing IDE 101) is provided in file `hw4_part2_output_01.txt` (can be found inside the `hw04_files.zip` file).

We will test your code with the values from the example file as well as a range of other values, including invalid commands and invalid ZIP Codes. Test your code well and when you are sure that it works, please submit it as a file named **hw4_part2.py** to Submitty for Part 2 of the homework. Be sure to use the correct filename or Submitty will not be able to grade your submission. You do **not** have to submit any of the files we have provided.