CSCI 1100 — Computer Science 1 Homework 4
Loops and Lists

Overview

This homework is worth **100 points** total toward your overall homework grade (Part 1 is worth 40 points and Part 2 is worth 60 points). It is due Thursday, October 18, 2018 at 11:59:59 pm. As usual, there will be a mix of autograded points, instructor test case points, and TA graded style points. There are two parts to the homework, each to be submitted separately. All parts should be submitted by the deadline or your program will be considered late.

The homework submission server URL is below for your convenience:

```
https://submitty.cs.rpi.edu/index.php?semester=f18&course=csci1100
```

Your programs for each part for this homework should be named:

```
hw4_part1.py
hw4_part2.py
```

respectively. Each should be submitted separately.

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

You will need the utilities and data we provide in `hw04_files.zip`, so be sure to download that and unzip it into your directory for HW 4. Module `hw04_util.py` is written to help you read information from the file. You should be able to write these modules in about a week yourself. But, for now, you can simply use them. The other file contains the actual ZIP code data.

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

Part 1: The Lifeguard Optimal Angle Problem (40 pts)

In Homework 1 you created a solution to compute the time it takes for a lifeguard to reach a drowning swimmer. For this assignment, we will extend that concept to solve an optimization problem by creating two functions to compute the angle at which a lifeguard needs to run to minimize the time it takes to reach the swimmer. We will then calculate some simple statistics over a number of runs.

The geometry of the problem shown in Figure 1 is the same as in Homework 1.

This is a little bit of a different assignment. Rather than have you write a complete program, we are going to have you write a module with two functions that we will call to reach a solution. This means that you will need to pay particular attention to how you write your functions; both to the parameter lists and to the values returned. On the other hand, you are not required to have any user interface (UI) code (code that interacts with the user, like calls to `input()` and `print()`). We will also provide (in Submitty) a module called `lifeguard` in file `lifeguard.py` that has one function, `get_response_time()`. The function is an implementation of the lifeguard solution you wrote in Homework 1, Part 1. You will need to call this function in the code you write to get the time it takes for the lifeguard to reach the swimmer under different conditions.

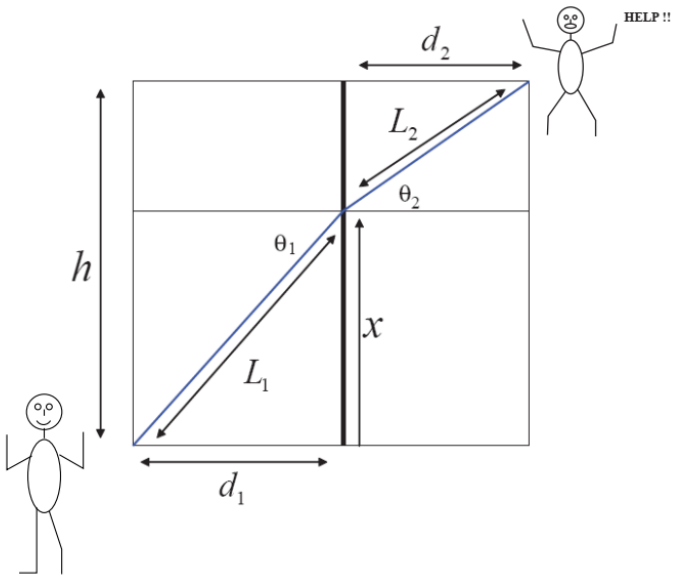

Figure 1: The lifeguard problem (picture credit: Massachusetts Institute of Technology, `https://ocw.mit.edu/courses/mechanical-engineering/2-71-optics-spring-2009/assignments/MIT2_71S09_usol1.pdf`)

See the notes section at the end of this handout for a discussion of testing, and for precise definitions of the inputs to and outputs from your functions.

Computing the optimal angle (30 pts out of 40)

Now start writing your code for `hw4_part1.py`. Create a function `get_optimal()`, that takes two tuples as its parameters. The first is a 6-tuple `beach` that contains the parameters from the lifeguard problem and the second parameter is a 3-tuple `interval` that contains a **start** angle, an **end** angle and a number of samples **num_samples**. The `get_optimal()` function needs to find the angle that reaches the swimmer in the shortest amount of time by checking **num_samples** angles beginning at the **start** angle and stopping at the **end** angle. This should be compared to the time the lifeguard actually took using the angle in the `beach` parameter. The function returns a 3-list with the following elements: the optimal time, the optimal angle and the difference between the time the lifeguard actually took and the optimal time.

An example run of `get_optimal()` (as executed in the Python shell) is provided below:

```
>>> import hw4_part1
>>> case = (8, 60, 40, 6, 1.2, 30.47)
>>> theta1_values = (0.0, 90.0, 10)
>>> print("Optimal: {}".format(hw4_part1.get_optimal(case, theta1_values)))
Optimal: [18.920333611703175, 60.0, 0.8392009413328374]
```

Collecting statistics (10 pts out of 40)

According to the American Red Cross, it is necessary "to ensure guards can respond to a water emergency, remove the victim and begin ventilations within $1\frac{1}{2}$ - 2 minutes within any part of any zone". For the purposes of our problem, we will assume that if a lifeguard reached a swimmer within 2 minutes (120 seconds), they were able to rescue the swimmer, if not then the swimmer drowned. The function `get_optimal()` lets us find out what happened on one beach scenario. Now we are going to run this for a large number of cases and collect some statistics to see if our lifeguards are well positioned on the beach and well trained. (Note: these are testing scenarios developed using a survey of the beach and quizzing our lifeguards. No actual swimmers were harmed in the making of this homework.)

Write a function `get_stats()`, that takes a list of an arbitrary number of 6-tuple `beach`es and the 3-tuple `interval` as its parameters and returns a 3-tuple `stats` with the following fields: `rescued`, `drowned_could_save` and `drowned_could_not_save`. These statistics contain information on how many swimmers lifeguards were able to rescue, how many swimmers were not rescued but could have been saved if a lifeguard had chosen a better angle to run towards the victim **(poor training)**, and how many swimmers were not rescued and there was no way a lifeguard could reach them within 120 seconds, no matter the angle **(poor positioning)**.

An example run of `get_stats()` (as executed in the Python shell) is provided below:

```
>>> import hw4_part1
>>> results = [(8, 60, 40, 6, 1.2, 30.47),
>>>            (8, 10, 50, 5, 2, 19.0987),
>>>            (18, 40, 20, 3, 1.5, 48.123),
>>>            (9, 10, 35, 5.5, 1.2, 45),
>>>            (17, 90, 150, 7, 1.1, 87.5),
>>>            (8, 12, 52, 6.5, 2.5, 29.0),
>>>            (8.9, 100, 100, 2.4, 3, 0.0),
>>>            (80, 52.5, 20, 4.5, 1.14, 78.55)
>>>           ]
>>> statistics = hw4_part1.get_stats(results, (0.0, 90.0, 1000))
>>> print("Rescued: {}; drowned and could save: {}; drowned and could not save: {}". \
>>>       format(statistics[0], statistics[1], statistics[2]))
Rescued: 5; drowned and could save: 2; drowned and could not save: 1
```

When you are sure your homework works properly, **name it correctly as hw4_part1.py** and submit it to Submitty. Your hw4_part1.py should contain definitions of two functions, `get_optimal()` and `get_stats()`. You may define additional functions, or even additional testing modules if you would like.

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 "to conduct research, develop web and mobile applications, design data visualizations" [1]. Among these is a dataset that provides geographic coordinates for U.S. 5 digit ZIP codes. For this assignment, we will be using a slightly improved version of the ZIP dataset available at [2]. 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.

The following commands are 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 a 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 => 12180
12180
ZIP code 12180 is in Troy, NY, Rensselaer 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

The utility module provided for this part of homework 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 hw04_util
zip_codes = hw04_util.read_zip_all()
print(zip_codes[0])
print(zip_codes[4108])
```

would give:

```
['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 in order: zip code (string), latitude (float degrees), longitude (float degrees), city (string), state (string), and county (string).

Implementation Details

You will need to define two functions that should strictly follow 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 written using any case or a mixture of lower and upper case.

   Return value:
   A list which contains 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.

E.g., `(42.673701, -73.608792, 'Troy', 'NY', 'Rensselaer')`

It is required that you implement functions according to specifications given above. 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.

Hints and Helps

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 `'\t'`. To generate the °, use `\xb0`. Just like the tab and the newline, this is a single character.
2. When printing out the location, there is a tab character at the start of the second line before `coordinates`.
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, 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 bath 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).

Sample Output

Below is an example output that demonstrates program features:

```
Command ('loc', 'zip', 'dist', 'end') => loc
loc
Enter a ZIP code to lookup => 12180
12180
ZIP code 12180 is in Troy, NY, Rensselaer county,
        coordinates: (042°40'25.32"N,073°36'31.65"W)

Command ('loc', 'zip', 'dist', 'end') => loc
loc
Enter a ZIP code to lookup => 1946
1946
Invalid or unknown ZIP code

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: (040°11'30.87"N,075°39'55.12"W)

Command ('loc', 'zip', 'dist', 'end') => loc
loc
Enter a ZIP code to lookup => 00000
00000
Invalid or unknown ZIP code

Command ('loc', 'zip', 'dist', 'end') => loc
loc
Enter a ZIP code to lookup => ksahdkja
ksahdkja
Invalid or unknown ZIP code

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

Command ('loc', 'zip', 'dist', 'end') => dist
dist
Enter the first ZIP code => 12180
12180
Enter the second ZIP code => 12180
12180
The distance between 12180 and 12180 is 0.00 miles

Command ('loc', 'zip', 'dist', 'end') => dist
dist
```

```
Enter the first ZIP code => 12180
12180
Enter the second ZIP code => 55499
55499
The distance between 12180 and 55499 cannot be determined

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

Command ('loc', 'zip', 'dist', 'end') => zip
zip
Enter a city name to lookup => Amsterdam
Amsterdam
Enter the state name to lookup => NL
NL
No ZIP code found for Amsterdam, NL

Command ('loc', 'zip', 'dist', 'end') => help
help
Invalid command, ignoring

Command ('loc', 'zip', 'dist', 'end') => END
END

Done
```

When you have tested your code, please submit it as `hw4_part2.py`. Be sure to use the correct filename or Submitty will not be able to grade your submission.

Some Notes on Part 1

Testing

When grading your work we will only call the two functions that you are required to implement. However, feel free to add code to the "main" level of your Python file that helps you test and debug your program or to define additional functions. Use proper program structure and remember that code you do not want to execute when your module is imported must be protected using the `if __name__ == "__main__":` construct we discussed in class. You are also **strongly** encouraged to use your Homework 1, Part 1 solution to write a "testing" version of our lifeguard module so that you can test your code prior to getting access to Submitty.

Writing the testing version is simple. Create a file, `lifeguard.py` and create within it a function `get_response_time(beach)` where the only parameter is the `beach` 6-tuple passed in to the `get_optimal()` function.

Embed the math from your Homework 1 Part 1 solution in this file without any input or output operations, using the appropriate fields from `beach` in place of any inputs. At the end, the single return value should be the time in seconds it takes to reach the swimmer.

This code is very useful for testing, but ultimately, it is throw away code. You do not need to submit this with your solution to Homework 4 Part 1, and if you do not quite have the right answer, remember, we will provide our own implementation of the `lifeguard` module on Submitty. Regardless of whether you write the module or not, you still need to `import lifeguard` and make call(s) to `get_response_time()` in your code for full credit.

The beach Definition

The 6-tuple `beach` has the following fields (all values are floats):

Index	Name	Units	Description
0	d_1	Yards	The shortest distance from the lifeguard to water
1	d_2	Feet	The shortest distance from the swimmer to the shore
2	h	Yards	Lateral displacement between the lifeguard and the swimmer
3	v_sand	Miles per hour	Lifeguard's running speed on sand
4	n	Dimensionless	Lifeguard's swimming slowdown factor
5	theta_1	Degrees	Direction of lifeguard's running on sand

The interval Definition

The 4-tuple `interval` has the following fields:

Index	Name	Data type	Units	Description
0	start	Float	Degrees	Starting value of `theta_1` for the parameter sweep
1	end	Float	Degrees	Ending value of `theta_1` for the parameter sweep
2	num_samples	Integer	Dimensionless	The number of samples taken for `theta_1` from the range $[start, end]$. Endpoints (`start` and `end`) are always included, i.e., `num_samples` cannot be less than 2. E.g., if `num_samples` is 3 it means that 3 angle values will be tried for `theta_1`: `start`, a value halfway between `start` and `end`, and `end`. Samples are taken at equal intervals between `start` and `end`.

The Return from get_optimal() Function

The function `get_optimal()` returns a 3-list with the following fields:

Index	Name	Data type	Units	Description
0	optimal_time	Float	Seconds	The shortest time a lifeguard can reach a swimmer
1	optimal_theta_1	Float	Degrees	The value of `theta_1` that corresponds to the `optimal_time`
0	actual_time	Float	Seconds	The time the lifeguard actually took based on theta_1

The Return from get_stats() Function

The function `get_stats()` returns a 3-tuple with the following fields:

Index	Name	Description
0	rescued	The number of swimmers successfully rescued (actual time no more than 2 minutes)
1	drowned_could_save	The number of swimmers not rescued when there was a chance to save them (actual time greater than 2 minutes, but optimal no more than 2 minutes)
2	drowned_could_not_save	The number of swimmers who could not be saved (optimal time more than 2 minutes)

References

[1] "Data.gov: The home of the U.S. Government's open data," 2018, Accessed on: Feb. 23, 2018. [Online]. Available: https://www.data.gov/

[2] "Download: Zip code latitude longitude city state county csv," San Diego, CA, USA, 2018, Accessed on: Feb. 23, 2018. [Online]. Available: https://www.gaslampmedia.com/download-zip-code-latitude-longitude-city-state-county-csv/