CSCI 1100 — Computer Science 1 Homework 3
Tuples, Lists, and If statements

Overview

This homework is worth **100 points** toward your overall homework grade, and is due Thursday, February 21, 2019 at 11:59:59 pm.

The goal of this assignment is to work with lists and tuples and use if statements. As your programs get longer, you will need to develop some strategies for testing your code. Here are a few simple ones: start testing early, and test small parts of your program by writing a little bit and testing. We will walk you through program construction in the homework description and provide some ideas for testing.

As always, make sure you follow the program structure guidelines. You will be graded on program correctness as well as good program structure. This includes comments. Minimally, we expect a brief docstring comment block at the start of the submission detailing the purpose and a brief summary (you may also include additional information like your name and date); and docstring comments for each function you define detailing the purpose, inputs, and expected return values. Additional comments have to accompany any complicated sections of your code.

Fair Warning About Excess Collaboration

Please remember to abide by the **Collaboration Policy** you were given last assignment. It remains in force for this and all assignments this semester. We will be using software that compares **all** submitted programs, looking for inappropriate similarities. This handles a wide variety of differences between programs, so that if you either (a) took someone else's program, modified it (or not), and submitted it as your own, (b) wrote a single program with one or more colleagues and submitted modified versions separately as your own work, or (c) submitted (perhaps slightly modified) software submitted in a previous year as your software, this software will mark these submissions as very similar. All of (a), (b), and (c) are beyond what is acceptable in this course — they are violations of the academic integrity policy. Furthermore, this type of copying will prevent you from learning how to solve problems and will hurt you in the long run. The more you write your own code, the more you learn.

Make sure that you have read the **Collaboration Policy** for acceptable levels of collaboration and so you know how you can protect yourself. The document can be found on the Course Materials page on **Submitty**. Penalties for excess collaboration can be as high as:

- 0 on the homework, and
- an additional overall 5% reduction on the semester grade.

Penalized students will also be prevented from dropping the course. More severe violations, such as stealing someone else's code, will lead to an automatic F in the course. A student caught in a second academic integrity violation will receive an automatic F.

By submitting your homework you are asserting that you both (a) understand the academic integrity policy and (b) have not violated it.

Finally, please note that this policy is in place for the small percentage of problems that will arise in this course. Students who follow the strategies outlined above and use common sense in doing so will not have any trouble with academic integrity.

Part 1: Time for a Change (40 pts)

Suppose you are given a list of coins in your cash register. You have a customer come to your register with \$1.00 and a purchase worth \$1.00 or less. You need to look in your cash register for the correct change and provide it to your customer.

Write a program to first ask for the name of a file containing the contents of your register and read from that file the list of all coins you have. Report the contents and then ask the user for the cost of the item they are purchasing. Assume they are paying with \$1.00 and calculate the change they should receive using the coins in your register. Use larger values first, beginning with \$0.50 and continuing through \$0.25, \$0.10, \$0.05, and finally \$0.01. At the end, either report the correct change, or print a message saying that you cannot give correct change and print out what you need to satisfy the request.

To solve this problem, you will first read from a file how many coins you currently have using the function provided in `hw3_util` as follows:

```
import hw3_util
coins = hw3_util.read_change('coins_01-03.txt')
print(coins)
```

If you execute this code with the above file, you will get the `coins` list below.

```
[50, 50, 50, 50, 50, 50, 25, 25, 10, 5, 1, 1]
```

A very easy way to solve this problem is to use the `count()` function of lists. For example, given the above list, `coins.count(50)` returns 6. You need to develop the code structure to decide how many of each coin you need based on the amount of change needed and the coins you actually have in your register.

Four examples of the program run (how it will look when you run it using Wing IDE 101) are provided in files `hw3_part1_output_01.txt`, `hw3_part1_output_02.txt`, `hw3_part1_output_03.txt`, and `hw3_part1_output_04.txt`. The first three examples make use of the coins file `coins_01-03.txt`. The fourth example uses `coins_04.txt`. (In order to access these files, you will need to download file `hw03_files.zip` from the `Course Materials` section of Submitty and unzip it into your directory for HW 3.)

Note that we only give you two coins test files. But, you must create other test files to make sure that your program works for all possible cases. Feel free to share your test files and test cases on the Forum. Discussing test cases on the Forum is a good way to understand the problem.

On the submission server, we will use different input files than the ones we provided in `hw03_files.zip`, so be ready to be tested thoroughly! We will also try a range of different input values. That said, this is not an exercise in combinatorics. Process the coins from largest down to smallest and do not worry about going back and running all combinations. As an example, review the output file `hw3_part1_output_04.txt` which uses coins file `coins_04.txt`.

Test your code well and when you are sure that it works, please submit it on Submitty as a file named **hw3_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. Note that you do not need loops for this part. Use the appropriate list functions. Use of loops will be penalized in the grading.

Part 2: Pikachu in the Wild! (40 pts)

Suppose you have a pikachu that is standing in the middle of an image, at coordinates `(75, 75)`. Assume the top left corner of the board is (0,0) like in an image.

We are going to walk a pikachu around the image looking for other pokemon. This is a type of simple simulation. First, we will set the parameters of the simulation by asking the user for the number of turns, **turns**, to run the simulation (starting at turn 1), the name, **name**, of your pikachu and how often, **often**, we run into another pokemon. At this point we enter a simulation loop (**while**). Your pikachu walks 5 steps per turn in one of `(N)orth`, `(S)outh`, `(E)ast` or `(W)est`. Every turn, ask the user for a direction for your pikachu to walk and move your pikachu in that direction. You should ignore directions other than N, S, E, W. Every **often** turns, you meet another pokemon. Ask the user for a type (**(G)round** or **(W)ater**). If it is a ground type, **'G'**, your pikachu loses. It turns and runs 10 steps in the direction opposite to the direction in which it was moving before it saw another pokemon. (If the last direction was not a valid direction, your pikachu doesn't move.) If it is a water type, **'W'**, your pikachu wins and takes 1 step forward. Anything else means you did not actually see another pokemon. Keep track of wins, losses, and `"No Pokemon"` in a list.

At the end of turn **turns** report on where your pikachu ended up and print out its record.

You must implement at least one function for this program:

`move_pokemon((row, column), direction, steps)`

that returns the next location of the pikachu as a `(row, column)` tuple. There is a fence along the boundary of the image. No coordinate can be less than 0 or greater than 150. 0 and 150 are allowed. Make sure your `move_pokemon()` function does not return positions outside of this range.

You can use the following code to test your `move_pokemon()` function. Feel free to write other functions if you want, but be sure to test them to make sure they work as expected!

```python
from hw3_part2 import move_pokemon
row = 15
column = 10
print(move_pokemon((row, column), 'n', 20)) # should print (0, 10)
print(move_pokemon((row, column), 'e', 20)) # should print (15, 30)
print(move_pokemon((row, column), 's', 20)) # should print (35, 10)
print(move_pokemon((row, column), 'w', 20)) # should print (15, 0)
row = 135
column = 140
print(move_pokemon((row, column), 'N', 20)) # should print (115, 140)
print(move_pokemon((row, column), 'E', 20)) # should print (135, 150)
print(move_pokemon((row, column), 'S', 20)) # should print (150, 140)
print(move_pokemon((row, column), 'W', 20)) # should print (135, 120)
```

Now, write some code that will call these functions for each command entered and update the location of the pikachu accordingly.

Two examples of the program run (how it will look when you run it using Wing IDE 101) are provided in files `hw3_part2_output_01.txt` and `hw3_part2_output_02.txt` (can be found inside the `hw03_files.zip` file). In `hw3_part2_output_01.txt`, note that `f` is an invalid direction, so it has no effect on the pikachu's state, and `r` is an invalid pokemon type which gets flagged as a `"No Pokemon"` in the results list.

We will test your code with the values from the example files as well as a range of other values. Test your code well and when you are sure that it works, please submit it as a file named **hw3_part2.py** to Submitty for Part 2 of the homework.

Part 3: Population Change — with Bears (20 pts)

You are going to write a program to compute a type of population balance problem similar to the bunnies and foxes you computed in Lab 3. This problem will have bears, berry fields, and tourists. We will just use the word berries to mean the area of the berry fields. We will count the number of bears and tourists, as well.

Bears need a lot of berries to survive and get ready for winter. So the area of berry fields is a very important part for their population. Berry fields in general spread over time, but if they are trampled too heavily by bears, then they may stop growing and may reduce in size. Tourists are the worst enemy of bears, often habituating them to humans and causing aggressive behavior. Sadly, this can lead to bears being killed to avoid risk to human life.

Here is how the population of each group is linked to one another from one year to the next. Suppose the variable `bears` stores the number of bears in a given year and `berries` stores the area of the berry fields.

- The number of `tourists` in a given year is determined as follows. If there are less than 4 or more than 15 bears, there are no tourists. It is either not interesting enough or too dangerous for them.

  In other cases, there are 10,000 tourists for each bear up to and including 10 and then 20,000 tourists for each additional bear. It is a great idea to write a function for computing tourists and test it separately.

- The number of `bears` and `berries` in the next year is determined by the following formulas given the population of bears, berries, and tourists in the given year:

  ```
  bears_next = berries/(50*(bears+1)) + bears*0.60 - (math.log(1+tourists,10)*0.1)
  berries_next = (berries*1.5) - (bears+1)*(berries/14) - \
      (math.log(1+tourists,10)*0.05)
  ```

  Remember none of these values can end up being negative. Negative values should be clipped to zero. Also, bears and tourists are integers. The log function is in the `math` module.

  You must write a function that takes as input the number of bears, berries, and tourists in a given year and returns the next year's bears population and berry field area as a tuple.

  ```
  >>> find_next(5, 1000, 40000)
  (5, 1071.1984678861438)
  ```

Then write the main program that reads two values, the current population of bears, and the area of berry fields. Your program then finds and prints the population of all three groups (bears, berries, and tourists) for the first year and another 9 years (10 years total). You must use a loop to do this. The output is formatted such that all values are printed in columns and are aligned to the left within each column. The width of each column is exactly 10 characters (padded with spaces, if necessary). All floating point values need to be printed with exactly one decimal place.

Once completed, your program should output: the smallest and largest values of the population of bears, berries, and tourists reached in your computation. These values should be output using the same formatting rules as for the population values for each of the years.

An example of the program run (how it will look when you run it using Wing IDE 101) is provided in file `hw3_part3_output_01.txt` (can be found inside the `hw03_files.zip` file). Note that the number of bears may go down to zero and then come back up. Why? Bears from neighboring areas can move in. The min and max values for each of bears, berries, and tourists may come from different years.

We will test your code with the values from the example file as well as a range of other values. Test your code well and when you are sure that it works, please submit it as a file named **hw3_part3.py** to Submitty for Part 3 of the homework.