CSCI 1100 — Computer Science 1 Homework 4
Lists, Loops and If statements

Overview

This homework is worth **90 points** toward your overall homework grade, and is due Thursday, October 19, 2017 at 11:59:59 pm. It has three parts, each is worth 45 points. You are welcome to choose any two of the parts of the homework, or get partial credit on all three. If you complete all three fully (135 total points), the surplus points will be used as bonus towards a past homework grade (bonus will be capped at 30 points max). Bonus credits will not apply towards a future homework grade.

The goal of this assignment is to work further with loops, lists and if statements. As your programs get longer, you will need to develop some strategies for testing. Here are a few simple ones: solve a simpler version of the problem first without all the special cases, or even solve a simpler and different problem. Work from that solution to the current one.

Start testing early, and test small parts of your program by writing a little bit and testing. Ask yourself what is a good input and what is the expected output first, instead of waiting for Submitty inputs. This will allow you to understand the problem better.

Remember to use office hours, especially earlier in the week, for conceptual questions.

As always, make sure you follow the program structure guidelines. You will be graded on program correctness as well as good program structure.

Remember, we will be continuing to test homeworks for similarity. So, follow our guidelines for the acceptable levels of collaboration on Piazza under resources.

Part 1: Landing a Rocket

For this part of the project, we are going to do a simple time simulation. Imagine you are landing a rocket on a planet. At each moment there are two main forces acting on the rocket: thrusters which push up the rocket but use fuel and gravity which pulls it down, both of these change the `altitude` and the `velocity` of the rocket. We will recalculate the current state of the rocket at each second. Here is what you need to do:

1. To start, use input statements to read from user the initial altitude (in meters), the gravitational constant, `g`, for the planet (in unit `meters/sec^2`), and an initial amount of fuel. All of these should be floats. Assume input is correct.
2. Start at time `0` and speed `0`.
3. Enter the simulation loop that iterates through time at increments of 1 and ends when altitude of the rocket reaches `0`. Altitude can never be less than `0`. Use lists to track rocket's speed, altitude and fuel consumption at each step. At each iteration of the loop do the following:
   (a) If the fuel is above zero, ask the user to input a thrust value, a float. Each unit of thrust uses one unit of fuel. If the user thrust is higher than the fuel, than use the full fuel value.

   If there is no fuel, thrust is zero. Do not ask for input.

(b) Based on the thrust, current speed, altitude and gravitation, calculate the new speed and new altitude for this time step. This is explained below.

(c) Print out the current state of the lander including time, altitude, and speed.

(d) Increment the time by one.

4. Once the altitude reaches `0`, exit the loop. If the speed when you exited the loop was more than `2.2`, you crashed. Otherwise, you did good. Write a message.

5. Finally, use your speed, altitude, and fuel use lists to get some statistics. Find the time when the maximum speed was reached and report on the altitude at that point. Find out how much fuel was consumed from the time of maximum speed and report this value. Report on any remaining fuel. Remember that the index of a value in a list can be found with the `list.index(value)` function. You can use `slicing` and `sum` to total up the fuel.

   If you had more than one time the max speed is reached, you can return the first one.

The next_step function

To compute the speed and altitude in the next time step, you must write a function `next_step(g, thrust, altitude, speed)` which returns a tuple with a new `altitude` and `speed` as follows.

Let's assume that each unit of thrust is equal to exactly the same magnitude of acceleration as `g`, but in the opposite direction.

So with `thrust=0` the rocket accelerates down at value `g`. Speed increases.

With `thrust=1`, there is acceleration is zero. Speed is unchanged.

With `thrust=2` the rocket accelerates up at `g`, so speed should decrease by `g` (creating a negative acceleration). Larger values of `thrust` accelerate upward even more, decreasing the speed.

Under this assumption, calculate the new speed first and then use that value to calculate the new altitude. Since our event simulates exactly one second per loop, this is easy.

```
new_speed = old_speed  + acceleration
new_altitude = old_altitude + new_speed
```

Return these values to end the function. Here are some examples.

```
>>> next_step(9.8, 0, 100, 0) # Acclerate down due to gravity
(90.2, 9.8)
>>> next_step(9.8, 1, 100, 0) # No acceleration, thrust and gravity cancel
(100.0, 0.0)
>>> next_step(9.8, 2, 100, 0) # Accelerate up
(109.8, -9.8)
```

Note that when `speed>0` the rocket is falling, and when `speed<0` the rocket is going up.

When printing float values, always print `2` decimal places. Do not round any values.

Grading

We are going to use multiple test cases to let you get as many points as possible for this part even if you have problems getting it all to work. Our first test case will be free fall. We will always enter

`0.0` for the thrust, so even if you do not have thrust working, your program will get the correct answer. Our second test case will add thrust, but we will guarantee that you do not run out of fuel. Our third test case and beyond are anything goes! Similarly, first get the loop component to work correctly and then add the statistics at the end to your code. Test slowly and add more complexity once you know something is working.

Examples of each of these are given below, but note that the actual values used in our testing will differ.

Free fall

```
Enter starting altitude (meters) => 100.0
100.0
Enter the gravitational acceleration (m/second^2) => 19.6
19.6
Enter the total units of fuel => 100.0
100.0
Time 0 - Altitude: 100.00, Speed: 0.00
Enter the thrust => 0.0
0.0
Time 1 - Altitude: 80.40, Speed: 19.60
Enter the thrust => 0.0
0.0
Time 2 - Altitude: 41.20, Speed: 39.20
Enter the thrust => 0.0
0.0
Time 3 - Altitude: 0.00, Speed: 58.80
Crashed ...
Kapow ...
... All astronauts are now shorter and you owe us a lander...

Maximum speed of 58.80 was reached at time 3 and an altitude of 0.00.
After that you used 0.00 units of fuel.
At the end you had 100.00 units of fuel left.
```

Guaranteed fuel

```
Enter starting altitude (meters) => 100.0
100.0
Enter the gravitational acceleration (m/second^2) => 9.8
9.8
Enter the total units of fuel => 1000.0
1000.0
Time 0 - Altitude: 100.00, Speed: 0.00
Enter the thrust => 0
0.0
Time 1 - Altitude: 90.20, Speed: 9.80
Enter the thrust => 0
0.0
Time 2 - Altitude: 70.60, Speed: 19.60
Enter the thrust => 0
0.0
Time 3 - Altitude: 41.20, Speed: 29.40
Enter the thrust => 0
```

```
0.0
Time 4 - Altitude: 2.00, Speed: 39.20
Enter the thrust => 4.9
4.9
Time 5 - Altitude: 1.02, Speed: 0.98
Enter the thrust => 1
1.0
Time 6 - Altitude: 0.04, Speed: 0.98
Enter the thrust => 1
1.0
Time 7 - Altitude: 0.00, Speed: 0.98
Nice landing!
The world salutes you!

Maximum speed of 39.20 was reached at time 4 and an altitude of 2.00.
After that you used 6.90 units of fuel.
At the end you had 993.10 units of fuel left.
```

Anything goes

```
Enter starting altitude (meters) => 10.0
10.0
Enter the gravitational acceleration (m/second^2) => 9.8
9.8
Enter the total units of fuel => 5.0
5.0
Time 0 - Altitude: 10.00, Speed: 0.00
Enter the thrust => 100.0
100.0
Out of fuel, able to use thrust of 5.00
Time 1 - Altitude: 49.20, Speed: -39.20
Time 2 - Altitude: 78.60, Speed: -29.40
Time 3 - Altitude: 98.20, Speed: -19.60
Time 4 - Altitude: 108.00, Speed: -9.80
Time 5 - Altitude: 108.00, Speed: 0.00
Time 6 - Altitude: 98.20, Speed: 9.80
Time 7 - Altitude: 78.60, Speed: 19.60
Time 8 - Altitude: 49.20, Speed: 29.40
Time 9 - Altitude: 10.00, Speed: 39.20
Time 10 - Altitude: 0.00, Speed: 49.00
Crashed ...
Kapow ...
... All astronauts are now shorter and you owe us a lander...

Maximum speed of 49.00 was reached at time 10 and an altitude of 0.00.
After that you used 0.00 units of fuel.
At the end you had 0.00 units of fuel left.
```

When you have tested your code, please submit it as `hw4Part1.py`. You must use this filename, or Submitty will not be able to run and grade your code.

Part 2: Is this crazy password valid?

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

Rule 1: The password should be between 10 and 25 characters inclusive, starting with a letter.

Rule 2. The password should have at least one of the following characters `@ $` and no `%`.

Rule 3: 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 4: Each upper case character (if there are any) must be immediately followed by at least one underscore symbol (`_`).

Rule 5: Each numerical digit, if there are any, must be less than 4. (So, `a35` is not valid because of 5). Note that in Rule 3 you checked the existence of one at least number in 1-4 range, now you want to check that all numbers are on this range.

Write a program that asks the user for a potential password, then checks and prints whether it satisfies each of the above rules.

Then, if the password is a valid one (satisfying all the above rules), tell the user. If the password is not valid, but satisfies rule 1, suggest a valid password by taking the first 8 and the last 8 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, you print nothing.

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 1,2 and 3 do not require looping at all. For rule 4, the simplest solution involves simply looping through the string once. The same is true for rule 5. 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 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) >= 10
is_lowercase = word.islower()
if is_long_enough and is_lowercase:
   print ("It is long enough and lower case")
```

Here are some example runs of this program:

```
Enter a password => this_is_#lowercase
this_is_#lowercase
Rule 1 is satisfied
Rule 2 is not satisfied
Rule 3 is not satisfied
Rule 4 is satisfied
Rule 5 is satisfied
A suggested password is: this_is_42owercase
```

```
Enter a password => M_ixed@casinG
```

```
M_ixed@casinG
Rule 1 is satisfied
Rule 2 is satisfied
Rule 3 is satisfied
Rule 4 is not satisfied
Rule 5 is satisfied
A suggested password is: M_ixed@c42d@casinG
```

```
Enter a password => short1@%pass
short1@%pass
Rule 1 is satisfied
Rule 2 is not satisfied
Rule 3 is satisfied
Rule 4 is satisfied
Rule 5 is satisfied
A suggested password is: short1@%42t1@%pass
```

```
Enter a password => with2$valid3numbers
with2$valid3numbers
Rule 1 is satisfied
Rule 2 is satisfied
Rule 3 is satisfied
Rule 4 is satisfied
Rule 5 is satisfied
The password is valid
```

When you have tested your code, please submit it as `hw4Part2.py`. You must use this filename, or Submitty will not be able to run and grade your code.

Part 3: World-wide University Rankings

In this part, we will use data from `university_data.csv`, a file of world-wide university rankings with data for 1000 universities. The data comes from `https://www.kaggle.com/mylesoneill/world-university-rankings`.

To complete this part, you will import a module named `hw4_util` that works very similar to the `lab05_util` module we used in Lab 5. You will use this module to read in the `university_data.csv` file into a list of lists. Within this list of lists, the first item is a list of data field names (header), and the remaining entries are lists containing information about different universities (name at location zero). Here is a short example of how this module works using data for the first five fields from the header and two random rows:

```
import hw4_util

data = hw4_util.read_university_file("university_data.csv")

print (data[0][:5])  ##this is the header item in the list
print (data[400][:5])
print (data[800][:5])
```

Produces the following output:

```
['Institution', 'World rank', 'Country', 'National rank', 'Quality of education']
['Huazhong University of Science and Technology', 400, 'China', 16, 367]
['Showa University', 800, 'Japan', 59, 367]
```

Problem specification

Write a program that first reads the file as shown above into a list of lists. The program then asks the user for the name of a university and two indices from the list (between 1 and 1000). You can assume the entered indices are valid.

Search for the entered university name in the data. If the university is not found, provide an error message. If the university is found, print a table comparing it to the other two universities as shown below. This time, for simplicity, we will not TABs. Instead we will use the format function to pad the fields with spaces – we pad the first field to 25 characters and the remaining fields to 12 characters. You can easily accomplish this with the format function:

```
>>> "{0:12}".format("abc")   ## left justified
'abc         '
>>> "{0:>12}".format("abc")  ## right justified, this is what we use
'         abc'
```

Your program must include a function `find_university(data, uname)` that takes as input the list of lists you have read from the file and the university name you have read from the user. Your function should return the index of the university if it is in the data list or -1 if it is not found.

Here are sample outputs of this program:

```
University name => Rensselaer Polytechnic Institute
Rensselaer Polytechnic Institute
Line number for first university to compare (1-1000) => 120
120
Line number for second university to compare (1-1000) => 680
680
First university: Rice University
Second university: University of L'Aquila

                          First      Second       Yours
             World rank     120         680         282
                Country     USA       Italy         USA
        National rank        61          37         110
Quality of education         52         367         137
   Alumni employment        161         567         142
  Quality of faculty         95         218         218
        Publications        255         655         402
           Influence        150         678         346
           Citations         59         645         182
        Broad impact        130         590         292
             Patents         76         871         210
               Score      49.73       44.38       46.19
                Year       2015        2015        2015
```

A second run:

```
University name => Rensselear Polytechnic Institute
Rensselear Polytechnic Institute
Line number for first university to compare (1-1000) => 1
1
Line number for second university to compare (1-1000) => 10
10
University not found
```

When you have tested your code, please submit it as `hw4Part3.py`. Only submit your code, not the other files we already provided you. You must use this filename, or Submitty will not be able to run and grade your code.