CSCI 1100 — Computer Science 1 Homework 5
Loops, Lists and Files

Overview

This homework is worth **90 points** total toward your overall homework grade (each part is 30 points), and is due Thursday, October 22, 2015 at 11:59:59 pm. There are three parts to the homework, each to be submitted separately. All parts should be submitted by the deadline or your program will be considered late.

Note on grading. As mentioned in past homeworks, correctness and program structure will both be a part of your grade. All programs must have the:

```
if __name__ == '__main__':
```

line. Starting with this homework, there are no more utility files. You can now write your own utility functions! The homework submission server URL is below for your convenience:

`https://submit.cs.rpi.edu/index.php?course=csci1100`

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

```
hw5_part1.py
hw5_part2.py
hw5_part3.py
```

respectively. Each should be submitted separately.

Part 1: Bunnies and Foxes, equilibrium

As promised, we will revisit the bunny and fox example from Lab #3. Remember we had formula for computing the population of bunnies and foxes in the next year given the current population. Here is a slight variation of the functions from Lab # 3 combined into a single function that returns a tuple:

```
def next_pop(bpop, fpop):
    bpop_next = int(max(0,round((10*bpop)/(1+0.1*bpop) - 0.05*bpop*fpop)))
    fpop_next = int(max(0,round(0.4 * fpop + 0.02 * fpop * bpop)))
    return (bpop_next, fpop_next)
```

The population of bunnies and foxes change every year as a function of the above formula. What can we expect in the future? Option 1: bunnies or foxes will totally disappear and never appear again (convergence), option 2: the populations will remain unchanged after a certain point (convergence), or option 3: the population will never reach convergence. You will write a program in this part to check which one is expected to occur.

Your program will have the function `next_pop` given above. You must define a second function called:

`check_convergence(bpop, fpop)`

that takes as input the initial population of bunnies and foxes, and then computes the population of the following year continuously using a **WHILE LOOP**. Your loop stops and outputs the remaining population of bunnies and foxes until any one of the following is true:

(a) No bunnies or foxes (or both) are left, i.e. has population zero (convergence is true),

(b) The population of bunnies and foxes in the current year is identical to the previous year, i.e. population equilibrium is reached (convergence is true),

(c) None of the above is true, but you have computed exactly 100 generations (convergence is false).

Your function, when finished computing the while loop must return a 4-tuple:

`(bpop, fpop, iter, converged)`

where `bpop,fpop` are the final populations of bunnies and foxes, `iter` is the total number of iterations the while loop repeated to reach this number (i.e. number of times you called the `next_pop` function), and `converged` is either True or False based on whether convergence was achieved.

Your functions must not have any global variables and you **must use a while loop** in the convergence function.

Your program will use these functions by first asking the user for the current bunny and fox population. Assume the input is a valid integer.

Once a value is given for both populations, you must call function `check_convergence(bpop, fpop)` from your module and print out the returned values. Sample output from separate runs is given below:

```
Please enter the starting bunny population ==> 6
6
Please enter the starting fox population ==> 7
7
(Bunny, Fox): Start (6, 7) End: (30, 30), Converged: True in 8 generations
```

```
Please enter the starting bunny population ==> 10
10
Please enter the starting fox population ==> 10
10
(Bunny, Fox): Start (10, 10) End: (29, 30), Converged: False in 100 generations
```

```
Please enter the starting bunny population ==> 1000
1000
Please enter the starting fox population ==> 2
2
(Bunny, Fox): Start (1000, 2) End: (0, 41), Converged: True in 1 generations
```

When you are sure your program satisfy the requirements stated here, submit it.

When you have tested your code, please submit it as `hw5_part1.py`.

Part 2: Election Time in Pythonia

In the spirit of the budding election season, let us examine the current political struggles in the far-off land of *Pythonia*. You will run a fantasy election in which candidates are trying to win villages by obtaining more votes than their competitors. On election day, villagers head to the polls and cast their vote, and the results pour in state-by-state. Before the next state's outcome is announced, you decide to run a simulation to see which of two running candidates will be ranked higher based on the state's voting turnout and the candidates' current standing.

Write a program to read from the user a comma-separated string that represents the current standing of two candidates in the election. The input is of the form:

```
candidate name, states won, states tied, total representative votes
```

Compute the candidate's current standing (or "rank") by adding up the following:

- each state won is worth 5 points
- each state with a tied-win is worth 2 points
- count half (rounding down) of the total representative votes

One candidate is ranked higher than another simply if this score is higher than the other candidate's.

Now you prepare for the different possible outcomes of the next state's voting for the candidates. Each candidate can receive 0 to 4 representative votes from the villagers, so the possible outcomes look like 1-0, 1-1, 0-1, 0-0, 2-0, 2-1, 2-2, 1-3, etc.

In these outcomes, the candidate with more representative votes wins the state, and both candidates receive a "tied win" if the score is tied.

For any possible outcome, compute which candidate will have the higher standing after winning or losing and print the first letter of the winning candidate's name as the result, or '-' if the candidates then have the same standing. You will print this information in a 5 by 5 board.

For simplicity, assume the input will be in the correct format, with no spaces around commas.

You **must** use `for` loops to accomplish this. The output below uses 5 spaces for each column.

Here is a possible run of the program in Wing:

```
Enter candidate 1 stats ==> Lovelace,5,1,28
Lovelace,5,1,28
Enter candidate 2 stats ==> Turing,4,3,29
Turing,4,3,29
Columns are Lovelace's votes, rows are Turing's votes
Votes|  0    1    2    3    4
-----|------------------------
  0  |  L    L    L    L    L
  1  |  T    -    L    L    L
  2  |  T    T    L    L    L
  3  |  T    T    T    -    L
  4  |  T    T    T    T    L
```

In this case, Lovelace and Turing have two chances to be tied in rank if the outcome of this next state is 1-1 or 3-3. Here is another possible run:

```
Enter candidate 1 stats ==> Engelbart,6,0,32
Engelbart,6,0,32
Enter candidate 2 stats ==> Babbage,3,1,22
Babbage,3,1,22
Columns are Engelbart's votes, rows are Babbage's votes
Votes|  0    1    2    3    4
-----|------------------------
  0  |  E    E    E    E    E
  1  |  E    E    E    E    E
  2  |  E    E    E    E    E
  3  |  E    E    E    E    E
  4  |  E    E    E    E    E
```

In this case, Engelbart is already doing so well that even losing the state would not lower his rank below Babbage.

You must use functions for this part. Keep in mind as you break down this part into steps that a function should aim to solve just one task so that main code and other functions can call it whenever needed.

Here are some suggestions of functions you might want to write for this part:

- `make_board(candidate1, candidate2)`: return the board to print as a string (or list of strings)
- `print_board(board)`: print the board in the above format
- `winner(candidate1, candidate2, outcome)`: take two candidate's information and one possible outcome, and return the letter to print, the "winner"

The idea here is to do calculations in one area, and then use the result for printing in another. **Beware** that such a `winner` function must not change the original candidate's stats so that you may call the function multiple times (for each possible outcome). The rest is easily accomplished with a double `for` loop.

When you have tested your code, please submit it as `hw5_part2.py`.

Part 3: Baby Names

In this part, you will use real data from Social Security Administration on the number of times a certain name is given to a babies born in the United States. The information is given in a file. We provide you with two separate files, one is a much shorter file for testing (`babies_short.txt`) and the second one is a very long file with all the baby names given between years 2000-2010 (`babies.txt`).

Each line in the file looks like this:

```
Marnie,F,164
Darrow,M,17
```

This means that `Marnie` as a female name is given to 164 babies, and `Darrow` as a male name is given to 17 babies. Of course, it is possible for the some names to be given both to male and female babies.

Write a program that first reads the file name to be used for baby names and a cutoff `k` from the user. Your program then should find and print the top k most popular female baby names and the top k most popular male baby names (and for each name, the number of times it is given). You must put a new line after every third name to make your output look pleasant.

Here are possible expected outputs:

```
File name => babies_short.txt
babies_short.txt
How many names to display? => 2
2

Top female names
Elsa (3353) Marnie (164)
Top male names
Amilcar (238) Leshawn (157)
```

```
File name => babies.txt
babies.txt
How many names to display? => 5
5

Top female names
Emily (223590) Madison (193063) Emma (181156)
Olivia (155935) Hannah (155607)
Top male names
Jacob (273642) Michael (250370) Joshua (231771)
Matthew (221411) Daniel (203569)
```

There is a very simple trick to solving this homework. The sort function in Python can sort lists and lists of lists (or tuples). For example, suppose we sort a list of tuples containing 2 elements each:

```
>>> x = [ (2, 'a'), (6, 'b'), (4, 'c'), (6, 'a'), (1, 'f') ]
>>> x.sort()
>>> x
[(1, 'f'), (2, 'a'), (4, 'c'), (6, 'a'), (6, 'b')]
>>> x.sort(reverse=True)
>>> x
[(6, 'b'), (6, 'a'), (4, 'c'), (2, 'a'), (1, 'f')]
```

As we can see from the above example, when we sort x, it will first sort by the first value in each tuple. If the first value is equal, then it will sort by the second value. As a result, `(6, 'a')` comes before `(6, 'b')`. You can also sort in descending order by using `x.sort(reverse=True)`.

If you want to find the top names, you must create a list that you can sort first by the frequency of the name, then the name. In the unlikely case that two names have the same frequency, their ordering will be descending lexicographic order. We can fix them with a hack, but will skip that for now. You will learn very elegant ways to avoid this in your next class.

When you have tested your code, please submit it as `hw5_part3.py`.