CSCI 1133, Fall 2019
Programming Assignment 5
Due: 11:55pm, Wednesday October 9, 2019

Due Date: Submit your solutions to GitHub by 11:55 p.m., Wednesday, October 9th. We will do a pull from this time point. Do not upload anything to Canvas and PLEASE be sure to use proper naming conventions for the file, classes, and functions. We will NOT change anything to run it using our scripts.

Unlike the computer lab exercises, this is not a collaborative assignment. You must design, implement, and test your code on your own without the assistance of anyone other than the course instructor or TAs. In addition, you may not include solutions or portions of solutions obtained from any source other than those provided in class (so you are ONLY allowed to reuse examples from the textbook, lectures, or code you and your partner write to solve lab problems). Otherwise obtaining or providing solutions to any homework problem for this class is considered Academic Misconduct. See the syllabus and read section "Academic Dishonesty" for information concerning cheating. Always feel free to ask the instructor or the TAs if you are unsure of something. They will be more than glad to answer any questions that you have. We want you to be successful and learn so give us the chance to help you.

Instructions: This assignment consists of 2 problems, worth a total of 40 points. Solve the problems below by yourself, and put all functions in a single file called `hw5.py`. Use the signatures given for each class and function. We will be calling your functions with our test cases so you must use the information provided. If you have questions, ask!

Because homework files are submitted and tested electronically, the following are very important:

- You follow all naming conventions mentioned in this homework description.
- You submit the correct file, hw5.py, through Github by the due date deadline.
- You follow the example input and output formats shown.
- Regardless of how or where you develop your solutions, your programs should execute using the python3 command on CSELabs computers running the Linux operating system.

Push your work into Github under your own repo. The specific hosting directory should be: repo-<username>/hw5, where you replace <username> with your U of M user name. For instance, if your email address is bondx007@umn.edu, you should push your hw5 to this directory: repo-bondx007/hw5

The following will result in a score reduction equal to a percentage of the total possible points:

- Incorrectly named/submitted source file, functions, or classes (20%)
- Constraints not followed (40%)
- Failure to execute due to syntax errors (30%)

Use the following template for EVERY function that you write (note: a helper function is a function so it gets a template.)

```
#==========================================
# Purpose: (What does the function do?)
# Input Parameter(s): (Each parameter by name and what it represents)
# Return Value(s): (What gets returned? Possibilities?)
#==========================================
```

Problem A. (10 *points*) **Detecting Wizards**:

It is well documented that it is physically impossible for a college student to have enough time to get good grades, have a social life, and get enough sleep: at most two of the three goals can be achieved. Therefore, any student that does appear to have all three things must be using dark magic.

Write a function `wizards(grades, life, sleep)` that takes in three lists of student names (strings), where `grades` represents students who get good grades, `life` represents students who have a social life, and `sleep` represents students who get enough sleep. You can assume that all lists are in alphabetical order. The function must return a list of all students who are clearly wizards (because they appear in all three lists).

However, there is a complication. One of the wizards in this course really doesn't like how easy the Python `in` keyword makes everything, so they performed a magic ritual that will turn anyone using the `in` keyword for this problem into a newt. Therefore, **you must solve this problem without using `in`**.

Hints:

- You may assume that no name appears in a single list multiple times.
- There are two basic approaches to this: a complicated, but fast (in terms of runtime) method that uses only one loop by taking advantage of the fact that the lists must be in alphabetical order, and a simple, but slow method that uses three nested while loops. Either strategy is fine.
- If you're going with the complicated, efficient strategy, remember that you can compare strings alphabetically with operators like > and <=.
- If you're going with the triple while loop strategy, be careful about where you increment your loop variables inside the nested loop.

Constraints:

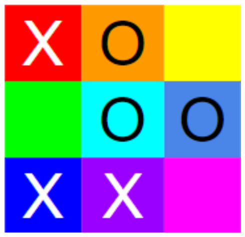
- Do not import/use any Python modules other than random (used for problem B).
- For this problem, you are not allowed to use the `in` keyword (this means no for loops).
- Don't use the input() function, as this will break our grading scripts.
- You are not allowed to use the Python set intersection, or anything similar, to trivialize the problem (don't worry if you don't know what that is at this point).
- Your submission should have no code outside of function definitions (comments are fine)

Examples:
```
>>> wizards(['Harry', 'Hermione'], ['Harry', 'Ron'], ['Hermione',
'Ron'])
[]

>>> wizards(['Aragorn', 'Frodo', 'Gandalf'], ['Aragorn', 'Gandalf',
'Gimli', 'Pippin'], ['Gandalf', 'Pippin'])
['Gandalf']

>>> wizards(['Zelda'], [], ['Ganondorf', 'Link', 'Zelda'])
[]

>>> wizards(['Mary', 'Spock', 'Sue'], ['Kirk', 'Mary', 'Sue'],
['Mary', 'Sue'])
['Mary', 'Sue']
```

Problem B. (30 *points*) **Playing Tic-Tac-Toe Randomly**

Tic-Tac-Toe is a very simple game involving a 3x3 grid, where players take turns, attempting to place 3 of their marks in a row, either vertically, horizontally, or diagonally (see the link for details if you are unclear on the rules). In this problem, you will be creating a program that plays tic-tac-toe by choosing an open spot at random, in order to determine how often ‘X’ wins, how often ‘O’ wins, and how often the game results in a draw when both players use that strategy.

We’ll be representing the board as a single list of 9 one-character strings in this problem. The first 3 elements of the list represent the top row of the board, the second 3 elements represent the middle row, and the last 3 elements represent the bottom row. Each string can either be 'X', 'O', or '-' (which represents an empty space). See the diagram below for an example of how to represent a tic-tac-toe board as a 9 element list.



```
['X', 'O', '-', '-', 'O', 'O', 'X', 'X', '-']
```

To do this, you need to write four functions. Be sure you match the names and arguments of the functions exactly to prevent grading issues.

`open_slots(board)` takes in a board list (formatted as specified above), and returns a list of the indexes which still contain a '-' (that is, the indexes of the open spots left). Indexes here refer to standard list indexing in Python, so since board is a 9 element list, they should all be numbers between 0 and 8, inclusive.

`winner(board)` takes in a board list (formatted as specified above), and returns a single character string that describes whether a player has won the game. In particular, it should return:

- `'X'`, if X won the game (there are three X’s in a row, horizontally, vertically or diagonally)
- `'O'`, if O won the game
- `'D'`, if the game ended in a draw (all spots are filled but neither player won)
- `'-'`, if the game is not over (there is at least one empty spot and neither player won)

You can ignore the possibility of getting a board where both X and O have achieved three in a row, since this should not happen in a real game.

`tic_tac_toe()` takes in no arguments, and simulates a single game of tic-tac-toe, returning a one character string that signifies the winner (`'X'`, `'O'`, or `'D'` for a draw). This simulation needs to follow the rules of tic-tac-toe: X and O alternate turns, starting with X; each player must choose an empty space on their turn; and the game ends as soon as someone wins or all spaces are filled. When choosing a space, the player should act randomly, with an equal chance to pick any of the remaining open spaces.

`play_games(n)` takes in one argument, an integer n representing the number of games to play. It then calls `tic_tac_toe` n times, and prints out the total number of times X won, the number of times O won, and the number of times that a draw occurred. This function does not return anything. Be sure to match the print formatting shown in the examples below.

You may also write any number of additional helper functions you think are useful.

Hints:

- Consider writing a function that displays a board list in the more traditional 3x3 grid form: this could be very helpful for debugging purposes.
- `(a == b == c == d)` works as you would expect (True if all four values are equal, False otherwise), so long as you don't separate any of the equality operations by parentheses.
- There are a LOT of different ways to write the `winner` function, and some are significantly more compact than others. Try to find an approach that's a bit more clever than 17 if statements.
- `random.choice` is a function that picks a random element from a list: this may be useful for choosing the next move from a list of open slots.
- You should probably do a lot more tests of `winner` than the four provided below; this will make your life easier when you get to the later functions.

Constraints:

- Do not import/use any Python modules other than random.
- You may use any list method that is appropriate to solving this problem.
- You can also use the `in` keyword again: the wizard from problem A has been defeated.
- Don't use the input() function, as this will break our grading scripts.
- Your submission should have no code outside of function definitions (comments are fine).

Examples:

```
>>> open_slots(['-', '-', '-', '-', '-', '-', '-', '-', '-'])
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> open_slots(['-', 'X', 'O', 'O', 'X', '-', '-', 'X', 'O'])
[0, 5, 6]
>>> open_slots(['O', 'X', 'O', 'X', 'X', 'O', 'X', 'O', 'X'])
[]
>>> open_slots(['X', 'X', 'O', '-', 'X', '-', 'X', 'O', 'O'])
[3, 5]

>>> winner(['X', 'X', 'O', 'O', 'X', 'X', 'O', 'X', 'O'])
'X'
>>> winner(['X', '-', 'O', 'X', 'O', '-', 'O', '-', 'X'])
'O'
>>> winner(['O', 'X', 'O', 'X', 'X', 'O', 'X', 'O', 'X'])
'D'
>>> winner(['X', 'X', 'O', '-', 'X', '-', 'X', 'O', 'O'])
'-'
```

(Note: the output of `tic_tac_toe` and `play_games` is random, so your results will not match the ones below. However, probability says that you should see X win roughly 58% of the time, O win about 29% of the time, and a draw about 13% of the time: if you're not getting close to that when running a large number of games then you may want to look for mistakes in your code).

```
>>> tic_tac_toe()
'X'
>>> tic_tac_toe()
'O'
>>> tic_tac_toe()
'D'

>>> play_games(1)
X wins: 1
O wins: 0
Draws: 0

>>> play_games(100)
X wins: 61
O wins: 30
Draws: 9

>>> play_games(10000)
X wins: 5917
O wins: 2798
Draws: 1285
```