CSE101 – Fall 2019
Programming Assignment #5

Due December 4, 2019 by 11:59pm, KST. The assignment is worth 25 points.

Instructions

For each of the following problems, create an error-free Python program.

- Download the Assignment 5 files and place them in your project directory.
- Submit Problem 1 as Assign5Answer1.py. For Problem 2, you will complete the provided game.py, board.py, and tile.py files and submit these files.
- These programs should execute properly in PyCharm using the setup we created in lab.
- At the top of every file add your name and Stony Brook email address in a comment.
- Please continue to use the naming conventions in Python and programming style that was mentioned in Assignment 1.

Problems

Problem 1: CSV File Processing (9 points)

A comma-separated values (CSV) file is a text file that uses a comma to separate values. A CSV file stores tabular data, such as numbers or text in a spreadsheet, in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. Thus, the file format gets its name from using the comma to separate different values.

Refer to the section on "Parsing CSV Files With Python's Built-in CSV Library" from the tutorial at this link: https://realpython.com/python-csv/#writing-csv-file-from-a-dictionary-with-csv to complete this problem.

A. (4 points) Using the code given in the section 'Reading CSV Files Into a Dictionary With csv' as an example, write your own function called `readcsv()` which reads in the CSV file at the path 'worldpopulation.csv' and prints the following output for each country:

```
GI Gibraltar has a population of 34733 in 2018, and will have a
population of 35897 in 2030.
TC Turks and Caicos Islands has a population of 35963 in 2018, and
will have a population of 41528 in 2030.
```

For reading the CSV file, please note that each row read by the reader is a dictionary, and you will need to update the tutorial code to read the dictionary with the appropriate keys for the worldpopulation.csv file. The keys are the names in the first row (e.g. "Flag", "Country") of the CSV file.

B. (5 points) Refer to the section called 'Writing CSV File From a Dictionary With csv'. Create your own function called `writecsv()` in which you will copy and extend the same code that you wrote for the `readcsv()` function to perform the following:
   a. Calculate the ratio of the 2030 population to the 2018 population for each country. For example, for South Korea, this ratio is 1.0300.
   b. Write this ratio along with the other parameter values to a new file named `'worldpopulationchange.csv'`. The first few lines in this file should look like the following:

Flag	Country	Population2018	Population2030	Ratio
CN	China	1415045928	1441181813	1.018469991
IN	India	1354051854	1512985207	1.11737612
US	United States	326766748	354711670	1.085519479
ID	Indonesia	266794980	295595234	1.10794901

For solving part b, one approach is before reading data in the worldpopulation.csv file, create an output CSV file and set its headers using the following code:

```
output_file = open('worldpopulation_growth.csv', mode='w')
fieldnames = ['Flag', 'Country', 'Population2018', 'Population2030',
'ratio']
output_writer = csv.DictWriter(output_file, fieldnames=fieldnames)
output_writer.writeheader()
```

Then when you are reading each dictionary `row`, you can create a new key 'ratio' and assign it a value using the following logic:

```
row['ratio'] = float(row["Population2030"])/ float(row["Population2018"])
```

Problem 2: Making a Minesweeper game (16 points)

For this problem, you will implement a text-based version of the Minesweeper game. Minesweeper is a single-player computer game in which you search tiles on a game board to "clear" areas, while trying to avoid exploding any of the hidden mines. You can mark suspected mines with "Flags" so you know where to avoid. You lose if you hit a mine, and win if you manage to successfully search and clear all of the tiles that do not contain a mine.

You can read more about Minesweeper at this Wikipedia link. You can also play a version of Minesweeper at this link.

This problem will give you practice using object-oriented programming with different classes. Make sure you have downloaded the Assignment 5 files, as this contains the three class files you will be using to build your Minesweeper game. These files contain all of the methods you will need to write the game, though you will need to write most of the functionality yourself. You are welcome to create new methods or modify any existing methods if you find it helpful, though this is not necessary to finish the game.

Your three classes are:

`Game`: a class that will contain the logic for playing the Minesweeper game, taking user input and updating the board with the input.

`Board`: a class that stores all of the game tiles and is able to generate the initial game board. It also has methods to search and "flag" tiles on the board.

`Tile`: a class that represents a single game tile. The tile keeps track of it's own state: whether it has a mine, has been searched, or has been flagged. It also keeps track of how many mines are nearby (that is, on tiles touching this tile). You can call print on a tile and it should display the appropriate description, given the tile's status (e.g. " " if not searched, "F" if flagged, "M" if searched and contains a mine, otherwise the number of nearby mines if it has been searched (ranging from 0 to 8) and doesn't contain a mine itself.

You will create this game over the course of several parts. **Complete each part and test it before moving on to the next part.**

Before you start, get familiar with the three class files you were given. Try running the program from the game.py file and see what happens. Right now, it prints out the instructions for how your game will work when you are finished and prints out a blank game board, with the rows and columns labeled with 1-8 and A-I, respectively.

Part 1: Implement handling user input to set a flag on a Tile object at the specified column and row

(3 points)

A tile in Minesweeper can be "flagged" if you suspect it has a mine on it. This will be displayed on the game board by the "F" character.

Your first task is to take user input that indicates flagging, a row, and a column, and then update the tile's status at the location to be flagged, and ensure the "F" symbol is displayed on that tile on the game board.

You will do this in multiple steps:

A. Write code in the Game class's `play_game` method to handle user input in the form of "flag b 5", which will "flag" the "b" column at row "5". You need to translate this user input into the correct column and row index for the tile object on the Board class. The user inputs the column as a letter, but this must be translated into the correct column index number. For example, if the user inputs "flag b 5", this means column index 1, row index 5.

   Assume that proper input always starts with the word "flag", then the column letter and row number, always in that order, and all separated by spaces. Any number from 1-8 or letter from A-I should be valid, and your program should accept upper or lower-case characters (e.g. "FLAG B 8" or "flag b 8"). If the input is not valid, print out "Invalid input" and ask the user for input again.

B. Write the code for the `get_tile_for_position` method in the Board class to get a tile at a specified column and row index. This method should return `None` if the index is not valid, and otherwise return the specified tile.

   Note that the tiles variable is a list that contains a list for each row, which can be accessed in the form of `tiles[row_index][column_index]`. It may be helpful to study the `print_board` method in the Board class to see examples of using the tiles list.

C. Next we want to write code to set a tile as "flagged" when the user enters a valid column and row index and have the flagged tile display an "F" when the board is printed.

   To do this, you will need to fill out the `flag_position` method in the Board class and update your code in the `play_game` method to call `flag_position` when a tile should be flagged. `flag_position` should update a Boolean instance variable in the Tile class to represent the flagged status.

   Lastly, you will need to modify the `__repr__` method in the Tile class to adjust how the tile is printed on the board.

D. Then, update the display for the user by calling `print_board` again after you get user input in the `play_game` method. At this point, test that your program is working as expected.

   For example, my sample output is below:

```
>>> Enter a position to search or flag a mine: flag b 6
   A B C D E F G H I
0:
1:
2:
3:
4:
5:
6:   F
7:
8:
```

E. Finally update `flag_position` method that marks a tile as "flagged" to mark a tile as *not flagged* if the user tries to flag the tile a second time. For example, If I ran "flag b 6" twice, I would have a blank board again.

Part 2: Generate mines and determine nearby mine counts

(5 points)

Now that you can flag tiles, you need to create mines on the board. In Minesweeper, a set number of mines are placed at the start of the game randomly around the board. You need to place these mines, and then for all of the tiles that do not contain a mine, calculate and store how many of the 8 adjacent tiles contain a mine.

Let's do this in multiple steps:

A. First, we will randomly place the mines in the Board class. The Board class's initializer takes in three arguments: the width of the board, the height of the board, and the number of mines. In the Board's initializer method, write code to add the provided number of mines randomly to the tiles. You can use Python's random library, or write your own randomization code for this. Be sure not to place a mine on a spot that already contains a mine.

   Just like storing the tile's "flagged" state, you will also want to store whether a tile has a mine or not as an instance variable.

   You can test your program by changing the `__repr__` method in the Tile class to display an "M" for all tiles that have a mine. A sample output of 10 randomly placed mines is below.

```
   A B C D E F G H I
0:       M
1:               M
2:     M M
3:
4: M M           M   M
5:     M         M
6:
7:
8:
```

B. Next, we need to determine each tile's nearby mine count. We will look at the 8 adjacent tiles for determining the count (or fewer than 8 tiles if the tile is at the edge of the Board).

   Write code to loop through every tile and determine the number of adjacent mines. Store this information on the tile.

   Hint: it can be helpful to use your previously made method to get a tile at a specified row and column index, which returns `None` if the tile does not exist. This can be an easy way to deal with edges of the board.

   To test your program, change the `__repr__` method in the Tile class to display either a mine if it has a mine, otherwise the nearby mine count. My sample output for this is below:

```
   A B C D E F G H I
0: M M 2 M M 1 0 1 1
```

```
1: 2 3 3 3 2 1 0 1 M
2: 0 1 M 1 0 0 0 1 1
3: 0 1 1 1 0 1 1 1 0
4: 0 0 0 0 1 2 M 2 1
5: 0 0 0 0 2 M 4 M 1
6: 0 0 0 0 2 M 3 1 1
7: 0 0 0 0 1 1 1 0 0
8: 0 0 0 0 0 0 0 0 0
```

Part 3: Take user input to search a tile

(3 points)

The next step is to handle user input to “search” a tile. This will be similar to marking a tile as “flagged”, but user input will be in the format "e 3" or "a 6". Note there is no word “flag” in front of the search input. We then need to update the display and if the user has “searched” a tile with a mine, end the game.

A. First, fill in the `search_position` method on the Board class that takes in a column and row index to search, and retrieves the appropriate tile and marks that tile as searched.

   To implement the `search_position` method, you will need to add another instance variable to the Tile class to track if it has been “searched” or not, just like for you tracked whether a tile was “flagged”, and update this “searched” variable after the user has searched the tile.

   You can also use the return value of `search_position` to signal whether the search found a mine or was successful in searching a tile without a mine.

B. Handle the user input to search a tile in the Game class and call the `search_position` method. After a tile on the board is searched, print the board again.

   You will need to update the Tile `__repr__` method to now only display a mine or nearby mine count **IF** the tile has been searched, otherwise show the blank " " string that represents an unexplored space.

   If the user searches a tile with a mine, print a message saying they lost and end the program. Otherwise, continue asking the user for their next position to search.

   My sample output is below where it searches first a tile that does not contain a mine, before searching and finding a mine.

```
   A B C D E F G H I
0:
1:
2:
3:
4:
5:
6:
```

```
7:
8:

>>> Enter a position to search or flag a mine: e 5
   A B C D E F G H I
0:
1:
2:
3:
4:
5:         3
6:
7:
8:

>>> Enter a position to search or flag a mine: e 4
   A B C D E F G H I
0:
1:
2:
3:
4:         M
5:         3
6:
7:
8:

You hit a mine and lose.
```

Part 4: Recursively search nearby safe tiles

(3 points)

In the game of minesweeper, when the user chooses to search a tile that has 0 mines in nearby tiles, the game knows it is safe to additionally search all 8 of those nearby tiles, and does this automatically so the game is faster to play. We will now implement that functionality

A. In your `search_position` method, if the tile that was searched has 0 nearby mines, write code to call `search_position` again for all 8 neighboring tiles that have **NOT** been searched already.

   Note that you should keep doing this as long as you keep finding tiles that have 0 mines in nearby tiles.

   You can test that this is working by adjusting the game to only have 1 or 2 mines, and then choosing to "search" a new space. The board display should update to show almost all tiles as "0"s for 0 nearby mines.

B. Sample output of recursively searching tiles is displayed below:

```
   A B C D E F G H I
0:
1:
2:
3:
4:
5:
6:
7:
8:

>>> Enter a position to search or flag a mine: b 5
   A B C D E F G H I
0:
1:
2:
3:
4: 1 1 1
5: 0 0 1
6: 0 0 1
7: 1 1 1
8:
```

Part 5: Winning the game

(2 points)

Almost done! Now you just need to check if the user has won. In Minesweeper, the user wins if they have successfully searched every Tile that does NOT contain a mine.

A. Write code to check if the user has won after each round of user input. If the user has won, print a message saying they have won and end the program.

   Here is a sample output for the end of a game that is eventually won. Please note that this is testing the game with only 3 mines, which can make it easier to test. Also note that you do not need to flag every tile with a mine to win, you only must search every tile that does not contain a mine.

```
>>> Enter a position to search or flag a mine: i 7
   A B C D E F G H I
0: 0 0 0 0 0 1   1 0
1: 0 0 0 0 0 1 1 1 0
2: 0 0 0 0 0 0 1 1 1
3: 0 0 0 0 0 0 1   1
4: 0 0 0 0 0 0 1 1 1
5: 0 0 0 0 0 0 0 0 0
6: 0 0 0 0 0 0 1 1 1
7: 0 0 0 0 0 0 1 F 1
8: 0 0 0 0 0 0 1
```

```
>>> Enter a position to search or flag a mine: i 8
   A B C D E F G H I
0: 0 0 0 0 0 1   1 0
1: 0 0 0 0 0 1 1 1 0
2: 0 0 0 0 0 0 1 1 1
3: 0 0 0 0 0 0 1   1
4: 0 0 0 0 0 0 1 1 1
5: 0 0 0 0 0 0 0 0 0
6: 0 0 0 0 0 0 1 1 1
7: 0 0 0 0 0 0 1 F 1
8: 0 0 0 0 0 0 1   1

>>> Enter a position to search or flag a mine: h 8
   A B C D E F G H I
0: 0 0 0 0 0 1   1 0
1: 0 0 0 0 0 1 1 1 0
2: 0 0 0 0 0 0 1 1 1
3: 0 0 0 0 0 0 1   1
4: 0 0 0 0 0 0 1 1 1
5: 0 0 0 0 0 0 0 0 0
6: 0 0 0 0 0 0 1 1 1
7: 0 0 0 0 0 0 1 F 1
8: 0 0 0 0 0 0 1 1 1

Congratulations, you won!
```

Congratulations! You've just built your own version of the Minesweeper game.