Just two graded problems this week, but you can do and submit the 3rd for extra credit! Include docstrings for *all* defined functions please! Do not forget to adjust the README to indicate you have completed the assignment before your final commit.

Get Assignment link: https://classroom.github.com/a/G5T4r3wl

1. (7 points) In the repository is a text file called `hands.txt` which has 1000 randomly dealt sets of poker hands to two players. Each line of the file contains 10 pairs of characters which indicate the card number and suit dealt to the players. The first 5 sets belong to the first player and the second 5 sets belong to the second. The first character of each set provides the card values (2–9,T,J,Q,K,A) while the second provides the suit (H,C,D,S). So the character pair `TH` would correspond to the 10 of Hearts for instance.

   (a) The template for a predicate function entitled `is_flush` has been provided for you. The goal of this function is to take a *single* player's poker hand and tell you if that player has a flush or not, where a flush is if every card has the same suit. Fill out the function so that it returns `True` if the hand is a flush and `False` if not. Calling the function individually would look like:

   ```
   >>> is_flush("TH 4H AH JH 8H")
   True
   >>> is_flush("TH 4D AS JH TS")
   False
   ```

   (b) Now write a function called `count_flushes` which takes a string filename as input, reads in the text file corresponding to that filename, and counts the total number of flush containing hands in the file. In addition to returning this value, the function should create a new text file with the same name as the original but with `_flushes` appended before the `.txt`. So that if the input filename was `hands.txt`, the output filename would be `hands_flushes.txt` This new file should contains a listing of every *individual* hand that was a flush, each on a separate line. You can test this function with the provided `hands.txt` file. Make sure you upload the output file as well as your code to Github!

2. (7 points) In the last several decades, a logic puzzle called *Sudoku* has become popular throughout the world. In Sudoku, you start with a 9×9 array of integers in which some of the cells have been filled with a digit between 1 and 9 as shown in Figure 1a. The challenge in the puzzle is to fill each of the empty spaces with a digit between 1 and 9 so that each digit appears exactly once in each row, in each column, and in each of the smaller 3×3 squares. The solution appears in Figure 1b. Each Sudoku puzzle is crafted so that there is only a single solution.

   Although the algorithmic strategies you need to computationally solve a Sudoku puzzle are beyond the scope of this class, we *can* write a function to check to see whether a proposed solution follows the Sudoku rules against duplicating values in a row, column, or outlined 3×3 square. Here you will write a function called `is_valid_puzzle` that takes a 9×9 array of integers and returns `True` if that array obeys all the Sudoku rules. It can be *very* useful here to write several helper functions to assist you along the way, so think about how you might decompose this problem into simpler pieces!

		2	4		5	8		
	4	1	8				2	
6				7			3	9
2				3			9	6
		9	6		7	1		
1	7			5				3
9	6			8				1
	2				9	5	6	
		8	3		6	9		

(a) The starting state of a sudoku puzzle. All empty spaces need to be filled in with a digit between 1 and 9.

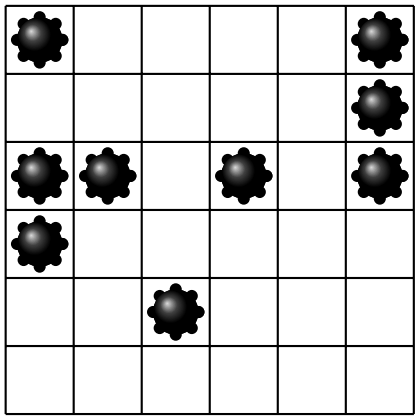
3	9	2	4	6	5	8	1	7
7	4	1	8	9	3	6	2	5
6	8	5	2	7	1	4	3	9
2	5	4	1	3	8	7	9	6
8	3	9	6	2	7	1	5	4
1	7	6	9	5	4	2	8	3
9	6	7	5	8	2	3	4	1
4	2	3	7	1	9	5	6	8
5	1	8	3	4	6	9	7	2

(b) A finished sudoku puzzle, with all spaces filled in. This is a valid solution in this case, as each row, column, and bold 3×3 square has each number appearing only once.

Figure 1: Examples of the starting and ending states of a sudoku puzzle.

To help you test your program, I have included a text file with 10 potential puzzle solutions and a function which reads them all in for you. The first puzzle matches the above figure. The first 5 puzzles are all valid, and the last 5 all have issues. Make sure your program returns the correct boolean for each!

3. (6 points (bonus)) In the game Minesweeper, a player searches for hidden mines on a rectangular grid that might, for a very small board, look something like:



One way to represent that grid in Python is to use a list of Boolean values marking the mine locations, where `True` indicates the location of a mine. You could, for example, initialize the variable `mine_locations` to the above 2D array by writing the following:

```
mine_locations = [
    [ True  , False , False , False , False , True  ],
    [ False , False , False , False , False , True  ],
    [ True  , True  , False , True  , False , True  ],
    [ True  , False , False , False , False , False ],
    [ False , False , True  , False , False , False ],
    [ False , False , False , False , False , False ]
]
```

Your task in this problem is to write a function called `count_mines` that takes a two-dimensional Boolean array (of any size) representing the location of the mines and returns a new array with the same dimensions that indicates how many mines are in the neighborhood of each location. If a location contains a mine, the corresponding entry in the matrix returned by `count_mines` should be `-1`. Thus, for our above example

```
counts = count_mines(mine_locations)
```

should set `counts` as follows:

```
[ [ -1 ,  1 ,  0 ,  0 , 2 , -1 ],
  [  3 ,  3 ,  2 ,  1 , 4 , -1 ],
  [ -1 , -1 ,  2 , -1 , 3 , -1 ],
  [ -1 ,  4 ,  3 ,  2 , 2 ,  1 ],
  [  1 ,  2 , -1 ,  1 , 0 ,  0 ],
  [  0 ,  1 ,  1 ,  1 , 0 ,  0 ] ]
```

Note that the neighborhood of each space includes those diagonal to the space. Edge spaces do *not* wrap around to the other side. Enroute to solving this, define a function called `check_index_location` which takes a row and column index, as well as the boolean list of mine locations, and returns the number of mines seen at that location. This function will be where most of the complexity lies, as you need to be careful to not ask for indexes that are out-of-bounds when on the edge pieces. As an example of output, the testing the function `check_index_location` might look something like:

```
>>> check_index_location(row=0, col=0, locs=mine_locations)
-1
>>> check_index_location(row=3, col=1, locs=mine_locations)
4
```

Once you have this function working properly, writing `count_mines` should be pretty straightforward.