Introduction: Part 1 of 4

Minesweeper is a fun computer game in which a player must avoid landing on a bomb while searching through a grid of covered spaces. In this project, we will implement our own version of Minesweeper so that you can play the game you wrote yourself!

Helpful terminology:

A board cell is a single location/space on the board.
A cell has four *neighbors*: in the north, east, south, and west.
A cell has eight *adjacent* cells: in the north, northeast, east, southeast, south, southwest, west, and northwest.

NW	North	NE
West	Me	East
SW	South	SE

The game:

In each round, a player clicks on a covered board cell.

If the clicked cell contains a bomb, the player loses the game. :(

Otherwise, the clicked cell is revealed, showing the number of adjacent cells that have a bomb. If none of the adjacent cells contain a bomb, then the neighboring cells are revealed as well! This continues until the entire area emanating out from the clicked cell is revealed, with numbered cells (i.e. cells that have at least one adjacent cell with a bomb) on the perimeter.

To get a feel for the game, you can search online to play a few rounds. Google has a good version of it that you can play by following this link.

Setting Up the Game

In this section, we'll give an overview of how we'll implement the game. There is no need to start implementing things quite yet, as we will go into more detail in the next Parts of this final project.
We will begin implementing Minesweeper by writing three classes: a `Cell` class, a `Board` class, and a `MinesweeperGame` class.

Each of these three classes has different responsibilities and *dependencies*. The `Cell` class is only responsible for keeping the state of a single cell in the game, such as whether it has a bomb or has been uncovered. A `Cell` object has no idea about its neighbors, its location on the board, or even that a larger board exists at all – because of this, the `Cell` class has **no** dependencies and can be written independently of the other two classes. The starter code for the `Cell` class is in the **minesweeper_final_project.py** file.

The `Board` class is responsible for holding the state of the board at any given point in the game, as well as providing methods of interacting with the board. Here, we will represent the actual board using a 2D *array*

(a list of lists) of `Cell` objects. Therefore, the Board is only dependent on the `Cell` class. (There are many different way we could have implemented the board – in fact, one of the best parts of using classes is our ability to change the internal representation without affecting the rest of the program!) The template for the `Board` class is also in the **minesweeper_final_project.py** file.

> *Stop and think*: Can you think of another way we could have implemented the board internally?

Last but not the least, the `MinesweeperGame` class will handle the mechanics of the game itself: making a `Board` object, running each round with user input, and finishing the game with the appropriate win/lose message. Because the `MinesweeperGame` class is dependent on the `Board` class, we will implement the `MinesweeperGame` class last. A template is available in the **minesweeper_final_project.py** file, but we won't implement this until later in the project.

Finally, we note that at the bottom of the `minesweeper_final_project.py` file, under where it says `''Begin here!''`, we have some sample code for how to run the game. For now, leave this commented out, but we will uncomment it in future classes when we have enough code to do a test run. Additionally, eventually your Minesweeper gameplay will progress from a console-based game to a graphics window as we add more features.

> *Stop and think*: We implemented our Tic-Tac-Toe game using only functions. This type of programming paradigm is called **procedural programming**. For the Minesweeper game, we will use classes and make instances of those classes (i.e. objects): **Object-oriented programming**. What are the pros and cons of each? When might we prefer object-oriented programming over procedural programming?

Let's Begin!

Now we're ready to start building the game!

The `Cell` Class

Often, when building up a large, complex program, we want to think of starting off with the smallest possible piece of the puzzle. In the case of the Minesweeper game, let's start with programming the class that represents an individual cell on the board.

> *Stop and think*: What should an **individual** cell be *responsible* for? What should an individual cell *know* about the rest of the program?

For example, should an individual cell know its location on the board? Although that might seem like an obvious choice to make (of course a cell should know its own location!), oftentimes you should give an object as *little* information as possible to start. In the case of Minesweeper, our implementation will only have cells keeping state information, such as whether the cell contains a bomb, whether the cell is hidden or revealed, and how many nearby bombs there are. Only the Board itself will keep track of which cell is in which location.

So, let's jump right in!

In our implementation, each `Cell` object will have three state variables, defined in `__init__(self)`:

- `has_bomb` - a variable that records if the `Cell` object has a bomb or not
- `covered` - a variable that records if the `Cell` object is covered (hidden from the player)
- `nearby_bombs` - a variable that records how many of its *adjacent* cells contain bombs

Additionally, it is customary to add *getter* and *setter* methods for each state variable a class has. These are functions that return the value of each variable (*getter*) and functions that set the value of each variable (*setter*).

Now it's your turn:
Initialize the three state variables and make *getter* and *setter* methods for each of them.

Stop and think: Why do we need setters and getters?

Sometimes, programmers will restrict what can be set via the *setter* function. In the case of the `Cell` class, for example, once a cell is uncovered, it will not become covered again within the course of a single game. Can you modify your covered *setter* so that it only sets `covered` from `False` to `True`? (**Hint**: Imagine just calling `cellObject.uncover()`)

Pretty Printing

For now, our Minesweeper game will be played solely on the console. Because of that, we want to be able to easily print out the contents of a cell. Let's give that responsibility to the cell itself!

Write a function `to_string(self)` that returns the state of the `Cell` object in one character. For example, you might represent a covered cell as "-", an uncovered bomb cell as "B", and an uncovered empty cell as the number of nearby bombs it has.

Stop and think: Why might we be returning a string here instead of printing a string?

Congrats—You've finished the first class of the Minesweeper game. Try to make a few cells, changing its state via *setter* methods, and printing out its state to the screen to test your class. A test template has been provided in the code, under where it says `# Testing for Cell class`; change the method names if you've named yours differently.

What do you expect to be printed out? Does it match what is printed out? Testing after completing each "chunk" of code is a good way to make sure you're on the right track for success!

The `Board` Class

Before we dive into the `Board` class, it's important to ask yourself the same questions as you did for the `Cell` class about *responsibilities*:

Stop and think: Why should a `Board` object be *responsible* for? What should a `Board` object *know* about the rest of the program?

Let's start simple: The Board will be responsible for keeping state of the Board and updating the state of the Board in each round of the Minesweeper game. The Board won't know what round of the game we're in, and it won't be responsible for getting user input or making sure the game loop keeps running—we'll leave that to the `MinesweeperGame` class later on.

We'll start by coding the Board's `__init__(self)` function. You should modify the `__init__` function so that instead of just `self`, it will also take in two extra parameters: `num_rows` and `num_cols`. In our implementation, each `Board` object should have **three** instance variables:

- `board_array` - a variable that represents the board as a 2D *array*
- `num_rows` - a variable that records the number of rows

- `num_cols` - a variable that records the number of columns

The `board_array` will be the internal representation of the board itself. The array (or, the list of lists) will hold `Cell` objects. You can imagine it looking something like this:

	Column 0	Column 1	Column 2
Row 0	cell_object1	cell_object2	cell_object3
Row 1	cell_object4	cell_object5	cell_object6
Row 2	cell_object7	cell_object8	cell_object9

which will have the internal representation

```
board_array = [[cell_object1, cell_object2, cell_object3],
[cell_object4, cell_object5, cell_object6],
[cell_object7, cell_object8, cell_object9]]
```

Another way to think of this representation is:

```
board_array = [row0, row1, row2]
```

What does each row represent?

Recall that a new `Cell` object is instantiated with `cell_objectX = Cell()`.

Stop and think: Why is it helpful to think of the same board representation in so many different ways? When might each of the different *mental* representations be beneficial?

Go ahead and fill the `board_array` with cells!

(**Hint**: The `board_array` is made up of rows, and for each row, there will be a certain number of cells. This sounds like the job of a nested loop!)

If you are having trouble, think about how to fill only one row of the board array. How can we wrap that procedure in another loop to fill the board array with rows of cells?

Pretty Printing

Before we move on to adding functionality for manipulating the Board, let's first write a method to print out the current board. Visualizing the board will be helpful down the road for debugging.

There are many ways to go about this particular problem of printing to the console, but our plan of attack will be the following: First, we'll loop through each row in the board array.

```
def pretty_print(self):
    print()
    for row in self.board_array:
        # Do something with each cell in row
```

At each round of the loop, we have a `row` and want to print out each individual cell's string representation in the row (which we already have done—the `cell.to_string()` method!) followed by a pipe (the vertical

bar, "|" character). However, each *individual* `print` statement automatically adds a new line, which is a problem.

> *Stop and think*: How can we print out each cell's string representation in the current `row` *on a single line*?

> Go ahead and finish the `pretty_print()` method.

> **Hint**: We don't want to call the `print` statement every time we move on to a new cell. How can we keep track of all the individual cells' strings?

Try to make a new board and print it out! You can use the part of the code that says `Testing for Board class (Part 1)` to help you out.
You are now done with the initializer of the `Board` class!

Board Setup

Whew—we've gotten through quite a bit of setup! Let's spend the remaining time adding some helpful functions that will be useful later on for the Minesweeper game.

The first helper method we'll write is to check if a certain row and column are in bounds. This will be helpful when checking user input later on:

```
def is_in_bounds(self, row, column):
    # Fill in here
```

> Finish the `is_in_bounds()` method.

> **Hint**: Don't forget, programmers use zero indexing! What should `board.is_in_bounds(0, 0)` return? If the board is a 3x3, what should `board.is_in_bounds(3, 3)` return? Recall, the board has class variables `self.num_rows` and `self.num_cols` - these might come in handy here!

Next, we'll write a method `add_random_bombs()` for adding random bombs throughout the board.

Some useful reminders:

1. To access a specific cell in any array, call `array[row_num][col_num]`

   > *Stop and think*: How do you access the very first cell? Remember, in programming we use zero indexing! You can then call `Cell` methods directly: `array[row_num][col_num].contains_bomb()`

2. With Python's `random` library, you can get a random number between `a` and `b` (inclusive) by calling `random.randint(a, b)`. Don't forget to add `import random` at the top of the file!

The method should take in parameters for the number of bombs, an exception row, and an exception column (these will be the first move made by the user—that way, you should not place a bomb at the position `(except_row,except_col)` so that the user will never *start* on a bomb! However any other positions in that column or row are up for grabs). The method will place the given number of bombs in random places except for the cell location given by the exception row and exception column.

First, initialize a variable that will count the number of placed bombs so far. Grab a random row index and random column index. When are we able to place a new bomb in a cell?

Go ahead and add the `add_random_bombs()` method.

Spend some time thinking and working on this—it's not a trivial task!

Some hints:

- Remember zero indexing when choosing a random row and column index! This doesn't just apply to having row-0 and column-0. Consider, what is the maximum possible index for a row or a column? Is it `num_rows` or `num_cols`? Why or why not?
- When checking that the randomly chosen row and column does not match the exception row and column, should you use `and` or `or`? Why?

Finally, once you are finished with the above functions, let's write a final helper function for "clicking" on a row and column (for testing purposes). We'll call it `click_on(self,row,col)` For now, let's just uncover that row and column on the board if the position is in bounds and print an error message if not.

Go ahead and add the `click_on()` method.

Now test out your board using the tests under where it says `Testing for Board class`! See the comments for notes about what you should expect to see.

Checkoff - End of Part 1:
You should

- Build the `Cell` class
- Set up the `Board` class with the internal board representation
- Add functionality for putting random bombs on the board

Show a course staff member your working code to get the checkoff.

In Part 2 we will:

- Define some global states for the `Board` class
- Finish the mechanism for running a full round of Minesweeper (from a user clicking on a cell through revealing all the neary cells, and checking for the end of the game)

In Part 3 we will:

- Code up the `MinesweeperGame` class
- Play some Minesweeper!

In Part 4 we will:

- Experiment with some fun game extensions :)

Continuing on our Minesweeper project, we will:

- Define some global states for the `Board` class
- Finish the mechanism for running a full round of Minesweeper (from a user clicking on a cell through revealing all the nearby cells, and checking for the end of the game)

Let's continue working!

1 Global States

For a computer, it is often easiest to keep track of state using numbers: perhaps a game has state 0, 1, or 2. For us humans, in order to program the game mechanics, we need to know what the state represents. Perhaps 0 means "game over", 1 means "game in play", etc. To make this interface between numbers and their meanings easier, we will consider what global states we want to keep track of. That is, what possible states any object instantiation might be in, and what values that state maps to. These are called global states because the entire program itself knows about them (think of them as global settings for the whole game). These states must also be mutually exclusive! (This means that it should not be possible to be in both state 0 and state 1 at the same time.)

Let's start by defining three game states. Based on the explanation above think of three states of the game.

We can now use these states throughout the entire Minesweeper Python file. For example, at some point, we might check if some `state == WON` and then do something accordingly. Imagine how confusing it would get if we had to check `state == 0` (what does 0 mean again?) or if we had to check `state == "WON"` (was it "won", "WON", "Win"?). Instead, because we defined a global variable WON, Python will do all the error-checking for us, and we can still preserve human-readability.

> Go ahead and declare some global constants at the top that you can use to track game states.

Great! Let's now return to our `click_on` method from last time.

2 Game Mechanics

Let's start by simply laying out in plain English what the game mechanics will look like. Note that we have defined the 3 game states as WON, LOST, and IN_PLAY:

1. A user will "click" on a cell. The board will "learn about" this click via the `click_on` method.
2. We should probably check if that cell has a bomb! If that's the case, how would the game state change? If so, uncover the cell (so the user can see a bomb was there) and return the global variable corresponding to the resulting game state.
3. Otherwise, we should both uncover the clicked cell and reveal some more surrounding cells according to the Minesweeper game rules. We should probably leave the 'search and reveal' process to another method entirely — let's tentatively call this `search_and_reveal`, and we can worry about implementing it later on.
4. Once we're done with searching and revealing the surrounding cells, we should check what the game state is like. That also seems like a useful piece to leave to a method of its own that we can worry about implementing later — let's call this `won_game` for now.

Not so bad! We've glossed over some stuff in steps 3 and 4 (specifically what happens in `search_and_reveal` and `won_game`), but this is a great start for laying out the groundwork for a fully functioning `click_on` method.

Go ahead and implement the `click_on` method. For now, ignore `search_and_reveal` and `won_game`, just so Python doesn't complain. We'll come back to those later.

3 Checking for a Won Game

Let's get the easy stuff out of the way and start with `won_game`.

Stop and think: What does it mean for a game to be won? How can you go about checking that there are no more cells that are bombless and have yet to be uncovered?

Hint: Consider the following code snippet:

```
for row in self.board.array:
    for col in row:
        # check: Is the cell uncovered and does it have a bomb?
        # If bombless and covered, what does that mean for the state of the game?
# What should you return if all cells without a bomb are uncovered?
```

Stop and think: If we make it outside the nested for loop, there must be no more uncovered bombless cells to click on. Why must this statement be true? Why is having a return statement inside the nested loop necessary for this statement to be true?

Go ahead and implement `won_game`.

4 Searching and Revealing

Before jumping into some hairy game mechanics, it's often very helpful to first write pseudocode, or a hybrid of Python and regular English, to make sure all of the game nuances are understood before actually writing the code. The search and reveal pseudocode might look something like this:

```
def search_and_reveal(self, row, col):
    # First, count the number of nearby bombs to (row, col)
    # Next, set the cell's nearby bomb count to reflect the number of bombs
    # Now uncover the cell
    if #there are no adjacent bombs (i.e. nearby bomb count is 0),
        # reveal all the *neighboring* cells
        # search around *those* cells too
```

Not so bad, actually! However, the very first thing we have to do is count the number of nearby bombs... and that sounds like the job of another helper method. Let's tackle that first.

Add the following method header:

```
def nearby_bomb_count(self, row, col):
    #First calculate the number of nearby bombs
```

Recall our image of adjacent cells in the North, Northeast, East, Southeast, South, Southwest, West, and Northwest:

NW	North	NE
West	**Me**	East
SW	South	SE

In order to rotate through all the adjacent cells, we can take advantage of the following observation:

All of the adjacent cells differ by a current cell (labeled "Me" in the image) by + or –1 in the row and column. Let's work through this concretely: given that the Me cell is in location (row 1, column 1) – remember, zero indexing! – fill in the following locations:

Direction (row, column)
North(__, __)
NE(__, __)
East(__, __)
SE(__, __)
South(__, __)
SW(__, __)
West(__, __)
NW(__, __)
Now imagine the Me cell is at arbitrary location (row r, column c). How would you apply the above to determine the location of each of the other cells in the diagram?

Awesome: We've verified that the row and column of each neighboring location relative to a cell at (r, c) maps to a +1, -1, or 0 offset. We can actually use some more global variables to help us make this mapping even easier to use in our code. For example, we can store `N=(-1,0)`. You could even store the directions as a single list.

> Add these global variables to your file, right under your other three global variables. You should now have a total of 11 global variables.

How can we use these directions in our `nearby_bomb_count` method?

First, recall that a for loop can map over tuples:

```
for tup1, tup2 in [N,NE,E,SE,S,SW,W,NW]:
    print(tup1, tup2)
```

At each iteration of the loop, what does `tup1` hold? How about `tup2`? Guess, and then try running this for loop to test your intuition.

The key intuition here is that we can use these tuple values add offsets from the inputted row and col. That is, set a counter equal to zero before a loop over the direction, and at each iteration of the loop, if there is a bomb at the location nearby row and col in that direction, add one to the counter and return the counter at the end.

> Go ahead and write the method `nearby_bomb_count(self, row, col)`.

> **Hint**: Make sure the nearby location is in bounds before checking if a bomb exists in that location! (What if you are checking location (0, 0), which has a northern neighbor at (-1, 0)... checking `bomb_array[-1][0]` will throw an error!). You even already have a helper method for that: we wrote `is_in_bounds` in Part 1!

Test your `nearby_bomb_count` method! You can use the part of the code listed under `Testing for Board class (Part 2)` to do this. Because you don't know where the bombs will be randomly placed, you may need to run your tests a few times to make sure it really works.

5 Back to Searching and Revealing

We can now get started on the main mechanics of each Minesweeper round: searching and revealing around a clicked-on row and column! Remember, we want to continually reveal cells that have no nearby bombs, and reveal nearby cells to *that* cell that has no nearby bombs, until an area has been revealed such that only the perimeter cells have nearby bombs, but no bombs have been revealed.

Let's recall our summary:

```
def search_and_reveal(self, row, col):
    # First, count the number of nearby bombs to (row, col)
    # Next, set the cell's nearby bomb count to reflect the number of bombs
    # Now uncover the cell
    if #there are no adjacent bombs (i.e. nearby bomb count is 0),
        # reveal all the *neighboring* cells
        # search around *those* cells too
```

Great! We can now count the number of nearby bombs, set the cell's nearby count accordingly, and uncover the cell at (row, col). We can also check if there are no adjacent bombs by just checking that the bomb count is zero.

Since we know that there are no adjacent bombs, we can be sure that all the neighboring cells in the N, NE, NW, S, SE, E, SW, and W directions also do not have bombs, so those cells can be revealed and checked as well.

> *Stop and think*: Why is it important here that a neighbor cell is covered before searching around that cell? If we don't check for this condition, we might end up in an infinite loop: First "uncovering" a northern neighbor, then "uncovering" the northern neighbor's neighbors, which includes the original cell as the southern neighbor, and then uncovering the southern neighbor's (i.e. the original cell's) neighbors, which will take us back to the already uncovered northern neighbor, and on and on... whew! Instead, we only want to uncover and search around cells that are still covered, since we know that is unexplored territory.

> *Stop and think*: We want to search and reveal around the neighbor cell, checking for the condition that its nearby bomb count is zero. It's almost as if the nearby cell has been "clicked" on, except not by the user, and instead by the mechanics of the game. Do we have a method that already handles searching and revealing around a certain row and column? The answer is yes: `search_and_reveal`!

We can actually use recursion to call `search_and_reveal` *inside* itself. However, as we noted before, we have to be really careful to check that, before calling the function recursively, we make sure that the nearby row and column are not yet explored and not out of bounds. These two checks ensure that there won't be an infinite loop due to the recursion.

> *Stop and think*: Consider a **fully covered** 3x3 board with **no** bombs. What happens when a user clicks on the middle cell? What would have happened if the user clicked on one of the corner cells? Try to work through the `search_and_reveal` process in each of these scenarios.

> Go ahead and implement `search_and_reveal`. You might want to do this in sections. You're welcome to code it however you want, but if you're not sure how to get started, try following the though process/ordering presented here, working through each *Stop and think* and the given pseudocode.

Checkoff - End of Part 2:
By the end of this part, you should have:

- Added some helpful global states
- Written a `click_on` method, which relied on:
  - A `won_game` method to test for a won game
  - A `search_and_reveal` method to explore the area clicked on

Your `click_on` method should now be fully functioning! Remember, `click_on` returns the game state after the click proceeds, so you can actually get the state from the call to `click_on`.:

```
game_state = test_board.click_on(0,0)
# will return 0 (WON), 1 (LOST), or 2 (IN_PLAY)
```

Show a course staff member your working code to get the checkoff.

Great work! Next up, we'll put everything together using the `MinesweeperGame` class.

In this part we will implement the console output for the game!

6 Init

Let's give the `MinesweeperGame` class two class variables: `self.board` to hold an instance of the Board class, and `self.num_bombs`, as a reference to be used later.

7 User Input

We should also write a method of grabbing user input, to eventually allow players to "click" on a row and column by typing in the console. The user input should look something like this:

```
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |

enter a row > 1

enter a col > 1

| 0 | 0 | 1 | - | - | - | - | - |
| 1 | 2 | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |

enter a row > 4

enter a col > 5

| 0 | 0 | 1 | - | - | - | - | - |
| 1 | 2 | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | B | - | - |
| - | - | - | - | - | - | - | - |

Ouch -- You stepped on a bomb! Thanks for playing!
```

Here, notice that the user-inputted row and column are **1-indexed**, such that an input like (1, 1) ends up "clicking" on `board_array[0][0]` and (4, 5) ends up "clicking" on `board_array[3][4]`.

> *Stop and think*: What might be the advantage to using 1-indexed user input? Even though that makes our jobs as programmers harder, why might it make playing the game (or, the "user interface") nicer?

We have to be careful of out-of-bounds input, though!

```
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
```

```
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |

enter a row > 0

enter a col > 0

Out of Bounds! Try clicking again.

enter a row > 1

enter a col > 1

| 1 | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
| - | - | - | - | - | - | - | - |
```

Once we convert the user input to be 0-indexed, we can use our Board class's `is_in_bounds` method (via our `board` class variable) to check. In more Python-like terminology, we should first ask for a row and column. While the input row and column are out-of-bounds, we should print a message to the screen and continue asking for a row and column until they are no longer out-of-bounds. Finally, we should return the clicked row and clicked column as a tuple, return (`clicked_row`, `clicked_column`)

> Go ahead and implement a method to ask for user input, `ask_for_input`.

> Go ahead and test your input function by calling it directly and printing the results, which should be the 0-indexed version of the input. Make sure your method correctly handles out-of-bounds indices.

> **Hint**: If only one index, either the row or the column, are out-of-bounds, should you still accept this as input?

Here is an example of how you could test your function (the comments are the thing that should print out):

```
minesweeper = MinesweeperGame(ROWS, COLUMNS, NUM_BOMBS)
user_input = minesweeper.ask_for_input()
print(user_input)
#Enter a row > 2
#Enter a col > 3
#(1,2)

minesweeper.play_game() #should still do nothing
```

8 Play Game!

Let's write `play_game(self)`.

Because we're working with the console, we should always start by printing the board. Guess what: we already wrote a helper for this in the Board class!

> Go ahead and start off the `play_game` method by printing the board using its `pretty_print` method.

Recall that we need to ask for the user's first input ***before*** placing all the random bombs.

> *Stop and think*: Why do we do this? What if we *didn't* do this?

So, the `play_game` method should then get input and add random bombs on the board (none of them being on the location of the user's first move). Only after adding the random bombs, which are guaranteed *not* to be where the user first clicked, should you call `click_on` on the users clicked row and column. **Remember: what does `click_on` return?** This will be helpful for making sure the game actually ends when it should!

We want to keep getting user input until the `click_on` function returns either a WON or a LOST state. ***Hint:*** *Whenever the word "until" is used in English, it's often indicative of a while loop...*

> Implement the rest of the game sequence in `play_game`! Don't forget to the print the board before every round.

Once the game is no longer in play, the local game state variable will point to either WON or LOST. Print something to the screen to indicate to the user how they did. For example:

```
enter a row > 2

enter a col > 2

| 2 | - | - | - | - | - | - |
| - | B | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |

Ouch -- You stepped on a bomb! Thanks for playing!
```

You can test a WIN scenario by setting the number of bombs to zero:

```
enter a row > 4

enter a col > 4

| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

You win!!
```

> **Checkoff - End of Part 3**:
> By the end of this part, you should have a complete working Minesweeper game! You can play it in the interactive console. Testing edge cases and making sure we get the expected behavior is a major part of coding! Show a course staff member your working code to get the checkoff.

In the next part, we will complete the final step in this project, moving from the console-based game to a graphics window output.

In this part we will finish up Minesweeper! Open up your working Minesweeper Python file. A working starter for Part 4's work can also be found in **minesweeper_final_project.py**. This is an optional task, so try this one only after you finish the required Part 1, Part 2, and Part 3 tasks.

9 Creating a Display Window

Import **graphics.py** and set up a window that we can play in. Just like concentration and tic-tac-toe, this board will have individual cells based on the number of rows and columns that the user inputs. It is helpful to define variables for the width and height of each cell.

```
#define window
win = GraphWin("Minesweeper", [width], [height] )
```

Stop and think: Where should you place this command?
Hint: It should be before we call `minesweeper.play_game()`!

In your code, you should replace `[width]` and `[height]` with the width and height you want to use for your window.

Stop and think: Instead of hard-coding the width and height of the window, can you use global constants and the number of rows/columns to calculate the height and width?

Go ahead and add this to your code.

10 Graphical Input

Instead of typing the row and column number we want the user to be able to click on the board to play the game. To do this we need to modify several methods in the `MinesweeperGame` class.

1. `ask_for_input` – This method takes in the row and column value that the user types into the console and returns `clicked_row` and `clicked_column`. Write a new version of this method called `graphical_input_helper` that takes the user's mouse click and converts it to row and column location.

   (a) You might need these commands:

   ```
   p = win.getMouse()
   x = p.getX()
   y = p.getY()
   ```

   (b) When converting x,y coordinate locations to row and column locations, remember that row correlates to the y axes and column correlates to the x axes. This can be very easily confused! Using integer division can also be helpful.

   Go ahead and add this function to your code.

2. `click_on` – This method calls `ask_for_input` to obtain the `clicked_row` and `clicked_column` values. Write a version called `ask_for_graphical_input` that accepts the row and column data that is returned by the new `graphical_input_helper` instead.

Go ahead and add this function to your code.

3. Because we changed the `input_helper` to `graphical_input_helper` and `ask_for_input` to `ask_for_graphical_input`, we need to make corresponding changes where those original methods are called under

   (a) `play_game`

   (b) `anywhere else`

Go ahead and make these changes to your code.

11 Printing the Board

Now we have an empty window and a way to input data using mouse clicks. But we can't play the game without setting up the board. The program already prints the board as strings in the console using the `pretty_print()` method under the Board class. How can we convert this code into a graphical representation?

For each cell in the `board_array`, create a rectangle object and set the colors to whatever you like. For example, you can use grey for an uncovered cell and black for a covered cell. Remember that to draw each cell, you need to provide a location.

Stop and think: How can you use the row number and column number to get an x and y coordinate value for each `Point` parameter in the `Rectangle` class?

Stop and think: How should you store the rectangle objects so that you can change their color later? Could you store them in a way similar to the `Cell` objects?

Additionally, we want to display text in each cell showing the number of bombs nearby. The `pretty_print()` function does this in the console using the `cell.to_string()` method. Find a way to convert this string representation of a cell to text that can be displayed on the board. Again, you have to supply a location for each text object! Try to get the numbers displayed in the center of each cell.

Stop and think: Once a character is revealed on the board, is it ever hidden again? Based on this answer, do you need to store the `Text` object you create?

Go ahead and add this functionality to your code.

12 Win/Lose Message

As a finishing touch, modify the code under the `play_game()` method under the `MinesweeperGame` class to display a message on the board when the user wins, loses, or encounters an error. Have the message positioned in the center of the board and choose different colors (e.g. green for "You won!", red for "You lost" and orange for "ERROR")!

Go ahead and add this functionality to your code.

13 Test Your Code

Run your code to see how it behaves. Start off with a small number of rows, columns, and bombs to make this easier. The console is still printing the string version of the board. Check to see if this matches with what is

happening in the graphical window! If you want, you can remove this print functionality later, but for now it can be useful for debugging.

Congrats! You're done with the Minesweeper game! Show off your hard work, and see if you can win at your own creation!

14 Other extensions:

1. Can you include a button that the user can click on to reveal a hint? That is, when the user asks for a hint, the location of one of the bombs should be revealed without losing the game. We would need to keep track of all the bombs and their locations at the beginning of the game and make sure a different bomb is uncovered each time the user clicks on hint.
2. Reveal all the bombs when the user loses. But make the square that the user last clicked on red so that they know where they had made a mistake.

   Using the `time.sleep()` method can be useful to display this right before the window closes!

Submitting your project

After you've finished your Minesweeper project, log into your Canvas account, find the post for Minesweeper in Assignments, and submit all of the files that you created or edited for Minesweeper (a final version). After you turn in your assignment, you're all done! Congratulations!!!