Welcome to Battleship!

In this assignment, you will build a simplified, one-player version of the classic board game Battleship! In this version of the game, there will be a single ship hidden in a random location on a 5x5 grid. The player will have 4 guesses to try to sink the ship.

To build this game we will use our knowledge of lists, conditionals, and functions in Python.

Getting Our Feet Wet

The first thing we need to do is to set up the game board.

Instructions

1. Create a variable `board` and set it equal to an empty list.

Make a List

Good! Now we'll use a built-in Python function to generate our board, which we'll make into a 5 x 5 grid of all `"O"`s, for "ocean."

```
>>> print (["O"] * 5)
```

will print out `['O', 'O', 'O', 'O', 'O']`, which is the basis for a row of our board. We'll do this five times to make five rows.

Instructions

1. Create a 5 x 5 grid initialized to all 'O's and store it in `board`.
    - Use `range()` to loop `5` times.
    - Inside the loop, `.append()` a list containing 5 `"O"`s to `board`, just like in the example above.

Note that these are the capital letter "O" and not zeros.

Check it Twice

Throughout our game, we'll want to print the game board so that the player can see which locations they have already guessed. Regularly printing the board will also help us debug our program.

The easiest way to print the board would be to have Python display it for us using the `print` command. Let's give that a try and see what the results look like—is this a useful way to print our board for Battleship?

Instructions

1. Use the `print` command to display the contents of the `board` list.

Custom Print

Now we can see the contents of our list, but clearly it would be easier to play the game if we could print the board out like a grid with each row on its own line.

We can use the fact that our board is a list of lists to help us do this. Let's set up a `for` loop to go through each of the elements in the outer list (each of which is a row of our board) and print them.

Instructions

1. First, delete your existing `print` statement.

   Then, define a function named `print_board` with a single argument, `board_in`.

   Inside the function, write a `for` loop to iterates through each `row` in `board` and `print` it to the screen.

   Call your function with `board` to make sure it works.

Printing Pretty

We're getting pretty close to a playable board, but wouldn't it be nice to get rid of those quote marks and commas? We're storing our data as a list, but the player doesn't need to know that!

```
>>> letters = ['a', 'b', 'c', 'd']
>>> print (" ".join(letters))
>>> print ("---".join(letters))
```

1. In the example above, we create a list called `letters`.
2. Then, we print `a b c d`. The `.join` method uses the string to combine the items in the list.
3. Finally, we print `a---b---c---d`. We are calling the `.join` function on the `"---"` string.

We want to turn each row into `"O O O O O"`.

Instructions

1. Inside your function, inside your `for` loop, use `" "` as the separator to `.join` the elements of each `row`.

Hide...

Now, let's hide our battleship in a random location on the board.

Since we have a 2-dimensional list, we'll use two variables to store the ship's location, `ship_row` and `ship_col`.

```
>>> from random import randint
>>> coin = randint(0, 1)
>>> dice = randint(1, 6)
```

1. In the above example, we first import the `randint(low, high)` function from the `random` module.
2. Then, we generate either zero or one and store it in `coin`.
3. Finally, we generate a number from one to six inclusive.

Let's generate a `random_row` and `random_col` from zero to four!

Instructions

1. Define two new functions, `random_row` and `random_col`, that each takes `board_in` as input. These functions should `return` a random row index and a random column index from your board, respectively. Use `randint(0, len(board_in) - 1)`. Call each function on `board`.

...and Seek!

For now, let's store coordinates for the ship in the variables `ship_row` and `ship_col`. Now you have a hidden battleship in your ocean! Let's write the code to allow the player to guess where it is.

```
>>> number = input("Enter a number: ")
>>> if int(number) == 0:
>>>....print ("You entered 0")
```

`input` asks the user for input and returns it as a string. But we're going to want to use integers for our guesses! To do this, we'll wrap the `input`s with `int()` to convert the string to an integer.

Instructions

1. Create a new variable called `guess_row` and set it to `int(input("Guess Row: "))`.
2. Create a new variable called `guess_col` and set it to `int(input("Guess Col: "))`.

It's Not Cheating—It's Debugging!

Now we have a hidden battleship and a guess from our player. In the next few steps, we'll check the user's guess to see if they are correct. While we're writing and debugging this part of the program, it will be helpful to know where that battleship is hidden. Let's add a `print` statement that displays the location of the hidden ship.

Of course, we'll remove this output when we're finished debugging since if we left it in, our game wouldn't be very challenging. :)

Instructions

1. Before the lines prompting the user for input:
    a. Print the value of `ship_row`.
    b. Print the value of `ship_col`.

You win!

Now we have the actual location of the ship and the player's guess so we can check to see if the player guessed right. For a guess to be right, `guess_col` should be equal to `ship_col` and `guess_row` should be equal to `ship_row`.

```
>>> if guess_col == 0 and guess_row == 0:
>>>....print( "Top-left corner.")
```

The example above is just a reminder about `if` statements.

Instructions

1. Add an `if` to check if `guess_row` equals `ship_row` and `guess_col` equals `ship_col`. If that is the case, please `print` out `"Congratulations! You sank my battleship!"` When you run this code, be sure to enter integer guesses in the panel where it asks for "Guess Row" and then "Guess Col".

Danger, Will Robinson!!

Of course, the player isn't going to guess right all the time, so we also need to handle the case where the guess is wrong.

```
>>> print (board[2][3])
```

The example above prints out `"O"`, the element in the 3rd row and 4th column.

Instructions

1. Add an `else` under the `if` we wrote in the previous step. Print out `"You missed my battleship!"` Set the list element at `guess_row`, `guess_col` to `"X"`. As the last line in your `else` statement, call `print_board(board)` again so you can see the `"X"`. Make sure to enter a col and row that is on the board!

Bad Aim

Now we can handle both correct and incorrect guesses from the user. But now let's think a little bit more about the "miss" condition.

1. They can enter a guess that's off the board.
2. They can guess a spot they've already guessed.
3. They can just miss the ship.

We'll add these tests inside our `else` condition. Let's build the first case now!

```
>>> if x not in range(8) or y not in range(3):
>>>....print "Outside the range"
```

The example above checks if either `x` or `y` is outside those ranges.

Instructions

1. Add a new `if` statement that is nested under the `else`. Like the example above, it should check if `guess_row` is not in `range(5)` or `guess_col` is not in `range(5)`. If that is the case, print out `"Oops, that's not even in the ocean."` After your new `if` statement, add an `else` that contains your existing handler for an incorrect guess. Don't forget to indent the code!

Not Again!

Now let's handle the second type of incorrect guess: the player guesses a location that was already guessed. How will we know that a location was previously guessed?

```
>>> print (board[guess_row][guess_col])
```

The example above will print an `'X'` if already guessed or an `'O'` otherwise.

Instructions

1. Add an `elif` to see if the guessed location already has an 'X' in it. If it has, `print ("You guessed that one already.")`

Test Run

Now you should have a game of Battleship! that is fully functional for one guess. Make sure you play it a couple of times and try different kinds of incorrect guesses. This is a great time to stop and do some serious debugging.

In the next step, we'll move on and look at how to give the user 4 guesses to find the battleship.

Instructions

1. Thoroughly test your game. Make sure you try a variety of different guesses and look for any errors in the syntax or logic of your program.

Play It, Sam

You can successfully make one guess in Battleship! But we'd like our game to allow the player to make up to 4 guesses before they lose.

```
>>> for turn in range(4):
>>>  # Make a guess
>>>  # Test that guess
```

We can use a `for` loop to iterate through a range. Each iteration will be a turn.

Instructions

1. Add a `for` loop that repeats the guessing and checking part of your game for 4 turns, like the example above. At the beginning of each iteration, `print "Turn", turn + 1` to let the player know what `turn` they are on. Indent everything that should be repeated.

Game Over

If someone runs out of guesses without winning right now, the game just exits. It would be nice to let them know why. Since we only want this message to display if the user guesses wrong on their last turn, we need to think carefully about where to put it.

1. We'll want to put it under the `else` that accounts for misses
2. We'll want to print the message no matter what the cause of the miss
3. Since our `turn` variable starts at 0 and goes to 3, we will want to end the game when `turn` equals `3`.

Instructions

1. Add an `if` statement that checks to see if the user is out of guesses.
    - Put it under the `else` that accounts for misses.
    - Put it after the `if/elif/else` statements that check for the reason the player missed. We want `"Game Over"` to print regardless of the reason.

    If `turn` equals `3`, `print "Game Over"`.

A Real Win

Almost there! We can play Battleship!, but you'll notice that when you win, if you haven't already guessed 4 times, the program asks you to enter another guess. What we'd rather have happened is for the program to end—it's no fun guessing if you know you've already sunk the Battleship!

We can use the `break` command to get out of a `for` loop.

Instructions

1. Add a `break` under the win condition to end the loop after a win.

To Your Battle Stations!

Congratulations! You have a fully functional Battleship game! Play it a couple of times and get your friends to try it out, too. (Don't forget to go back and remove the debugging output that gives away the location of the battleship!)

You may want to take some time to clean up and document your code as well.

Instructions

1. When you are done playing Battleship! and are ready to move on, click Run.

Acknowledgment
The assignment has been adapted from the Code Academy Python tutorial and modified to work in Python 3.