CS 120: Introduction to Computer Programming II

Battleship Long (Code)

due at 5pm, Wed 8 Dec 2021

1 Background

Battleship is a classic board game. Two people play against each other; each arranges their ships on a grid, and then take turns "shooting" at each other's board; one player announces a target, and the other player announces where that lands on their board: it may be a hit on one of the ships (perhaps sinking it), or a miss.

We are going to use this game to explore the interaction between multiple classes - and we are going to allow you to design the interface between them. I have provided the `main()` function for our program, and I'll also be providing a video, showing you the complete game in action. But it will be **your** job to implement the classes that my `main()` function is making use of.

Since you will be doing design for this assignment, we've broken it into two pieces. First, during a Short Problem, you will write up a design (expressed in English, not in code) for how your classes will work, and how they will communicate with each other internally. Your TA will review your design, and give you feedback on it.

Then, the Long Problem will be to implement Battleship. Note that we don't expect that your final design will exactly match your plan; changes are almost certain. But, we want you to spend time on the design, and that will (hopefully) make the actual implementation easier to accomplish.

2 What We Will Provide

For the design phase, we have provided a `main()` function for you. The code is available on the class website; examine it, to see how the program works, overall. (Later in this spec, we will also give you a formal definition of all of the classes and methods which your code must provide.)

3 What You Must Turn In

Design

For the design phase, you must turn in a simple design document to GradeScope. The file must be a PDF (you can generate a PDF using Google Docs, or with just about any word processor you have installed on your computer). The file must include a thoughtful design about how the various classes will interact with each other (what sorts of methods and attributes will they need), as well as the data structures that you will use inside them.

Obviously, we can't auto-grade this, and so your TA will be going over it by hand. Your job, in this document, is to think carefully through the issues that you will have to face when you write the code; the job of your TA is to give you feedback about how you could revise the design to make it better. Of course, your TA will have to assign a grade for your design, but this is not directly tied to your design; they might have lots of suggested changes to your design, and yet give you full credit. Or, they might not have much to say, and dock you significantly. This is because the purpose of this design document is to **demonstrate careful thought.** You're not getting graded on whether your design is "correct," or even if it is "good;" you're being graded based on whether it **demonstrates effort and care.**

See the short problem spec for some ideas about things you'll need to discuss in your document.

Code

For the implementation, you must turn in (at least) a single file, named `battleship.py`, which contains the required classes. We will be testing this part with an auto-grader.

4 New Python Feature: `assert`

In this project, you must use `assert` statements, inside your functions, to check for certain error conditions (see below for details). An `assert` looks like this:

```
assert thing_that_must_be_true
```

If the condition that you check is true (as expected), then an `assert` does nothing; your program simply continues. But, if the condition is not true, the `assert` will crash your program. (Technically, what happens is that your code throws an `AssertionError` exception.)

5 Program Overview

In a file `battleship.py`, you must write (at least) two classes, as follows:

- The class `Board`, which represents a single game board. Note that this is only **one half** of a game of Battleship; if we wanted to implement an actual, competitive game, then we would build two of these objects. But, for this little program, we're going to "play" with only one board. (Feel free to write a more advanced `main()` program, in your spare time.)
- The class `Ship`, which repesents a single ship in the game.

My code will call the following methods, and thus you must implement them exactly as defined (since I'll be auto-testing them). Feel free to use helpers

as appropriate; also, you almost certainly will need to implement some other methods, which I haven't detailed here.

5.1 `Board.__init__(self, size)`

This creates a `Board` object. The `Board` starts empty (no ships in it). All boards are square, so this function needs only a single size parameter.

Your code **must** assert that the size is positive.

5.2 `Board.add_ship(self, ship, position)`

This adds a ship to the board. The `ship` argument must contain some sort of information about the shape of the ship; the format is up to you, because **you** will be writing the code on both sides of the interface. The last argument is a `(x,y)` tuple, which gives the actual position of the ship on the board.

While the interface between the `Board` and the `Ship` is up to you, our `main()` function both creates the `Ship` object (doing rotation if necessary) and also calls `add_ship()`. So the end result - however it gets encoded in the data structure - must be predictable. For instance, suppose that `main()` creates a Destroyer (defined in `standard_ships.py`). The standard shap for a Destroyer is:

```
[ (0,0), (1,0) ]
```

meaning that it takes up two spaces. If you do not rotate the ship, then the first space `(0,0)` is on the left, and the second is immediately to the right of it.

Now, suppose that you call `add_ship()` to add this to a board. And the position where you add it is `(7,1)`. In that case, the ship will take up two spaces on the board: `(7,1)` and `(8,1)`.

Your code **must** use an assert to sanity-check that this new ship does not fall outside of the valid indices of the board, and also that it does not overlap any ship already added.

(spec continues on next page)

5.3 `Board.print(self)`

Prints out the current state of the board, to the screen. It should print out a grid something like this:

```
  +--------------------+
9 | . . . . . . . . . . |
8 | . B . . . S S S . . |
7 | . B . . . . . . . . |
6 | . B . . . . . . . . |
5 | . B . . A * A A A . |
4 | . . . o . . . . . . |
3 | . . . . . . . . . D |
2 | . X X X . . . . . D |
1 | . . . . . . . . . . |
0 | . . . . . . . . . . |
  +--------------------+
    0 1 2 3 4 5 6 7 8 9
```

You must support varying board sizes (1 and up). You can get **most** of the credit for this assignment while only supporting sizes up through 9 (that is, single digits). However, a few points (less than 5% of the total assignment) will be dedicated to testing larger boards, that require 2 digits. (We'll never test 3-digit sizes.) 2-digit sizes are drawn as follows:

```
   +------------------------+
11 | . . . . . . . . . . . . |
10 | . . . . . . . . . . . . |
 9 | . . . . . . . . . . . . |
 8 | . B . . . S S S . . . . |
 7 | . B . . . . . . . . . . |
 6 | . B . . . . . . . . . . |
 5 | . B . . A * A A A . . . |
 4 | . . . o . . . . . . . . |
 3 | . . . . . . . . . D . . |
 2 | . X X X . . . . . D . . |
 1 | . . . . . . . . . . . . |
 0 | . . . . . . . . . . . . |
   +------------------------+
                         1 1
     0 1 2 3 4 5 6 7 8 9 0 1
```

The grid squares (that is, the interior of the board) should print out according to the following table:

- Use period characters for empty spaces on the board, which have never been shot at.

- Use a lowercase oh (o) to represent a place, in deep water, which was shot at, but it was a miss - such as (3,4) in the above example.
- Use an asterisk (*) to record a place on a ship which has been **Hit,** but where the ship has not been sunk - such as (5,5) in the above example.
- Use all capital X's if a ship has been sunk - such as (1,2) through (3,2) in the above example.
- For parts of ships that have *not* been hit yet, that square must use the first letter of their name, so that it's easy to tell them apart. (Sometimes, two ships will have the same first letter. We just have to live with that.)

5.4 `Board.has_been_used(self, position)`

Checks to see if someone has already shot at this location; returns a Boolean value. In the above example, this would return True for (3,4), (5,5), (1,2) and other locations. It would return False for (1,8), (0,0), and many other examples.

This method must assert that the position given is on the board.

Like with `add_ship()` and the shape array for the `Ship` class, the position is an (x,y) tuple.

5.5 `Board.attempt_move(self, position)`

Handles a "shot" that is being taken at a given location on the board. The position parameter tells you where the shot is headed.

This method must use an **assert** to guarantee that the location is within the valid range of the board. It must also use an **assert** to verify that the location has never been shot at before.

This method must return a string, which explains what the result of the move was; the possible results are a miss, a hit (the ship is not named), or a hit (the ship name is included). See the testcases for the exact strings.

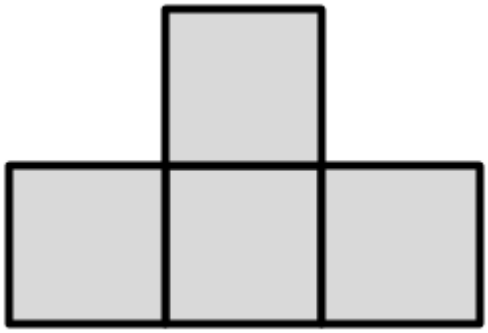
Like with `add_ship()` and the shape array for the `Ship` class, the position is an (x,y) tuple.

(spec continues on next page)

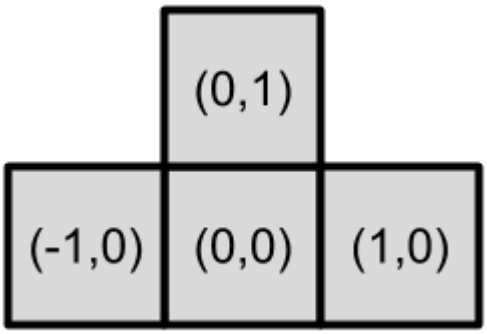
5.6 `Ship.__init__(self, name, shape)`

Builds a new `Ship` object. The first parameter is a string, which is the name of the ship; the second is the shape of the ship. The shape is given as a list of ordered (x,y) pairs. As noted in the `add_ship()` description above, these are **offsets** from a starting location; it would be normal (although it's **not required**) that the first element in any shape would be (0,0).

For example, imagine a "Tee" shaped piece from Tetris. (See `Tee()` in `tetris_ships.py`.) The shape is as follows:



We assign grid coordinates to these blocks. We could use any offsets at all, but in this example, we'll use (0,0) to represent the center block. Thus, the coordinates are:



We could encode this, as a shape parameter, as follows:

```
[ (0,0), (1,0), (0,1), (-1,0) ]
```

(spec continues on next page)

5.7 `Ship.print(self)`

Prints out a single line of output, which summarizes the state of the ship. The first several characters indicate whether the various parts of the ship have been hit; print out one character per element in the shape. If that space has been hit, print an asterisk; otherwise, print the first letter of the name. Then, print some blank spaces; print out exactly the number of spaces so that the shape-status, plus the spaces, totals to 10 characters. (You can assume that the shapes will never be more than 9 spaces long.)

So, for example, an Aircraft Carrier, which has been hit directly in the middle, would print out

```
AA*AA     Aircraft Carrier
```

NOTE: Unlike `board.print()`, where a ship turns to all X's when it has been sunk, `ship.print()` should always use asterisks for any hits it's taken. Thus, the Aircraft Carrier, if sunk, will print out

```
*****     Aircraft Carrier
```

NOTE: What counts as the 'first' or 'second' space in a ship, especially if it has a complex shape? Use the shape array: element `[0]` in the shape array is first, and so on.

5.7.1 How to Map Hits to the Printout

When a ship has been hit on one of its positions, what letter, of the printout above, should be marked with an asterix? Each letter in the printout represents one of the points in the shape of the ship: so, in the example above, an Aircraft Carrier has 5 points in it; apparently, point `[2]` of the shape array is the one that was hit.

Therefore, any time that there is a hit on a ship, you must figure out not only **what** ship was hit, but also what **part** of the ship - or rather, what part of the shape - was hit. Think about this when you design how the board and the ship will interact!

5.8 `Ship.is_sunk(self)`

Returns `True` if the Ship has been sunk (that is, if all of the slots in the ship have been hit; returns `False` otherwise.

5.9 `Ship.rotate(self, amount)`

This method is used to rotate a single ship. Rotate it around the (0,0) coordinate; thus, if one of the elements in the shape is (0,0) (as will be common), that one position should not change.

If the user has created many ships with the same shape, make sure that this method only rotates one of them.

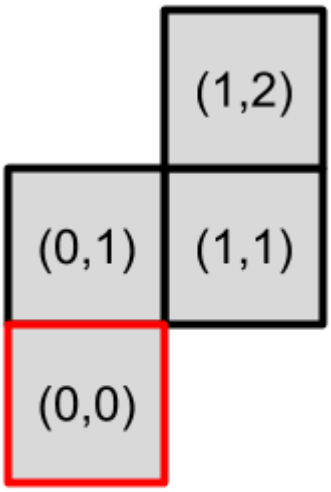
NOTE: This method **must** be called **before** the ship is added to the Board. You may assume that this will never be called after the ship has been added.

This updates the shape of the Ship, by creating a **new shape array** (don't re-use the old one!), where each coordinate pair has been modified. Each coordinate should be rorated in units of 90 degrees clockwise. That is, if `amount` is 0, don't change the shape; if it is 1, rotate clockwise by 90 degrees; if it is 2, rotate by 180 degrees; if it is 3, rotate clockwise by 270 degrees.

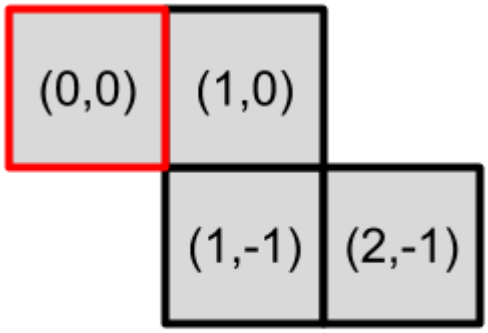
Your code must assert that the value of `amount` is between 0 and 3, inclusive.

To see how this works, consider the same shape `R_ZigZag` from `tetris_ships.py`, at all four rotation positions:

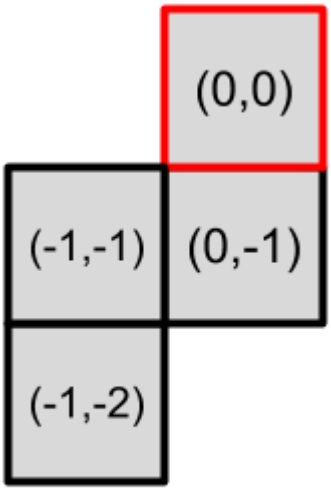
(spec continues on next page)



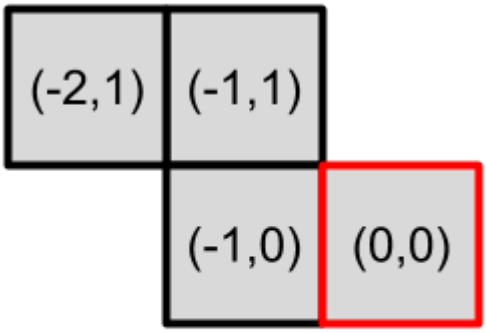
rot = 0



rot = 1



rot = 2



rot = 3

(spec continues on next page)

6 Turning in Your Solution

You must turn in your code using GradeScope. Turn in the following files:

- `battleship.py`

7 Acknowledgements

Thanks to Saumya Debray and Janalee O'Bagy for many resources that I used and adapted for this class.