Battleships

Chami Lamelas

August, 2021

Introduction

Battleships is a naval combat game played between two players. Players place a collection of ships of varying lengths on two dimensional grids. Then, the players take turns taking shots at their opponents ships. A player retains their turn as long as they are hitting enemy ships. Once they miss, it's their opponent's turn. The first player to sink all of their opponent's ships wins the game. A single ship takes up some k contiguous positions on the grid. A ship is sunk when all k positions on the ship have been hit.

A player cannot see their opponent's ships. However, they can keep track of their previous shots on a grid of their own. They can use their record to attempt to hit the opponent's ships. For this reason, it is safe to assume that a user will **not** try to fire at the same position multiple times. This is important as it will make writing some of the later functions easier.

Game Overview

When two players typically play, there are four boards.

- Player 1 Position Board : This is the board where player 1 positions their ships. When player 2 takes a shot, player 1 will use this board to tell them whether player 2 hit, miss, or sunk a ship. They will also use this board to track how much damage their ships have taken.
- Player 1 Record Board : This is the board where player 1 records their hits and misses on player 2's ships.
- Player 2 Position Board : This is the board where player 2 positions their ships. When player 1 takes a shot, player 2 will use this board to tell them whether player 1 hit, miss, or sunk a ship. They will also use this board to track how much damage their ships have taken.
- Player 2 Record Board : This is the board where player 2 records their hits and misses on player 1's ships.

In our implementation, we will track all of this information in two 2D lists (`player1_board` and `player2_board`) and two lists (`player1_ships` and `player2_ships`).

- `player1_ships` : This will track how much life remains in each of player 1's ships. This list will be initialized with the starting lengths of the available ship types.
- `player2_ships` : This will track how much life remains in each of player 2's ships. This list will be initialized with the starting lengths of the available ship types.

Thus, the lists of ships will be nonnegative integers.

Suppose that we start with n ships. Then, the entries of the board can take on one of $n+3$ values.

- `UNSEEN_WATER < 0`: This represents that the board's owner has not placed a ship here, but the opponent has not seen this yet.
- `SEEN_WATER < 0`: This represents that the board's owner has not placed a ship here and the opponent has discovered this by missing in some previous turn.
- `SEEN_SHIP < 0`: This represents that the board's owner has placed a ship here and the opponent has discovered this by hitting the ship here in some previous turn.
- $0, \ldots, n-1$: This will give the identifier of the ship that the board's owner has placed here but has not been discovered by the opponent.

We can now define our two boards.

- `player1_board`: This is a 2D list that represents the board owned by player 1. Player 1 will position their ships on this board.
  - `player1_board[i][j] == UNSEEN_WATER`: Player 1 has not placed a ship at (i, j) but player 2 has not discovered this.
  - `player1_board[i][j] == SEEN_WATER`: Player 1 has not placed a ship at (i, j) and player 2 has discovered this by missing in some previous turn.
  - `player1_board[i][j] == SEEN_SHIP`: Player 1 has placed a ship at (i, j) and player 2 has discovered this by hitting it in some previous turn.
  - `player1_board[i][j] == k >= 0`: Player 1 has placed a ship at (i, j) but player 2 has not discovered this. `k` is the identifier of the ship that was placed here. This identifier is just the index of the ship in `player1_ships`. Thus, if player 2 guesses this position, `player1_ships[k]` should be decreased by 1 to show that the ship has 1 less life.
- `player2_board`: This is a 2D list that represents the board owned by player 2. Player 2 will position their ships on this board.
  - `player2_board[i][j] == UNSEEN_WATER`: Player 2 has not placed a ship at (i, j) but player 1 has not discovered this.
  - `player2_board[i][j] == SEEN_WATER`: Player 2 has not placed a ship at (i, j) and player 1 has discovered this by missing in some previous turn.
  - `player2_board[i][j] == SEEN_SHIP`: Player 2 has placed a ship at (i, j) and player 1 has discovered this by hitting it in some previous turn.
  - `player2_board[i][j] == k >= 0`: Player 2 has placed a ship at (i, j) but player 1 has not discovered this. `k` is the identifier of the ship that was placed here. This identifier is just the index of the ship in `player2_ships`. Thus, if player 1 guesses this position, `player2_ships[k]` should be decreased by 1 to show that the ship has 1 less life.

Now, we also need to convey to player 2 how well they are doing on `player1_board` (and similarly how well player 1 is doing on `player2_board`). However, we cannot simply print these 2D lists to the console or the opponents will discover where their opponents ships are hiding. So far, our description of `player1_board`, `player2_board`, `player1_ships`, and `player2_ships` will serve as an implementation of the position boards described at the beginning of the Game Overview section. However, we will implement the record boards via a special 2D list print function that will just show to player 2 their hits and misses on player 1's board and vice versa.

Provided Files

In this section, I will discuss the three files provided in the starter code and how you should use them.

battleships_constants.py

This file contains a collection of constants you should be using throughout your code to avoid the use of magic numbers. **You should not edit this file.** Some of these constants will make more sense when you start doing the tasks.

- `UNSEEN_WATER`: This represents the board value you should use to signify unseen water as discussed above.
- `SEEN_WATER`: This represents the board value you should use to signify seen water as discussed above.
- `SEEN_SHIP`: This represents the board value you should use to signify a seen ship as discussed above.
- `MISS_MARKER`: When displaying player 1's progress on player 2's board, this is the character you should use to represent that player 1 has previously missed at this position (and similarly for player 2's progress on player 1's board).
- `HIT_MARKER`: When displaying player 1's progress on player 2's board, this is the character you should use to represent that player 1 has previously hit a ship at this position (and similarly for player 2's progress on player 1's board).
- `UNSEEN_MARKER`: When displaying player 1's progress on player 2's board, this is the character you should use to represent that player 1 has not fired at this position (and similarly for player 2's progress on player 1's board).
- `MISS_ID`: This signifies that a player's fire on a position missed.
- `HIT_ID`: This signifies that a player's fire on a position hit.
- `SUNK_ID`: This signifies that a player's fire on a position sunk a ship.
- `NORTH`: Players will be placing ships by specifying a position and then the direction it will point in. This signifies the north direction.
- `SOUTH`: This signifies the south direction.
- `EAST`: This signifies the east direction.
- `WEST`: This signifies the west direction.

battleships.py

In this file, you will be implementing a library of functions used for playing battleships. This includes creating and displaying boards (tasks 2.2,3.1, and 3.2), placing ships (tasks 2.1 and 4.1-4.4), and running and processing turns (tasks 5.1-5.3). You will notice that all the functions have a line `pass`. This is just a placeholder, you should remove it as you implement the functions.

two_player_battleships.py

This file will be your client code. You will be writing four functions in this file that are particular to preparing (tasks 7.1 and 7.2) and running (tasks 8.1 and 8.2) a 2 player game. Also, when running this file, you will be running a two player battleship game. For the purpose of testing your code, you will need to write your script (task 9) inside of the if statement

```
if __name__ == '__main__':
```

You can ignore this if statement and can run this file as normal.

1 Imports in battleships.py

In order to implement `battleships.py`, which file(s) do you need to import? Feel free to use aliases.

Hint: Remember, you will be writing functions to initialize boards, place ships, and process turns.

Related Reading: 10.2, 10.3

2 Set-up Functions

In this task, you will be writing the general battleship set up functions in `battleships.py`.

2.1 copy_list()

This function takes a list `ls` and returns a *deep copy* of the list.

Related Reading: 6.1, 6.3, 7.3

2.2 create_board()

This function takes a positive integer `size` and creates a 2D list with dimensions `size` by `size` initialized with `UNSEEN_WATER`.

Related Reading: 6.1, 6.3, 6.4

3 Displaying a Board

In this task, you will be writing the board display function in `battleships.py`.

3.1 get_display_symbol()

This function takes a 2D list `board` and tuple `pos` and returns the string *display symbol* that corresponds to the value of `board[pos[0]][pos[1]]`. Suppose that `board` is player 1's board. The display symbol corresponding to `board[pos[0]][pos[1]]` is what player 2 could have recorded on that position.

Hints: What value(s) of `board[pos[0]][pos[1]]` corresponds to a `HIT_MARKER`? `MISS_MARKER`? `UNSEEN_MARKER`?

Related Reading: 6.2, 6.4, 7.2

3.2 display_board()

This function takes a 2D list `board` and prints a display to the opponent. If `board` is player 1's board, we would `display_board(board)` to player 2. The displayed board must have the following form

```
 ? | ? | ?
---+---+---
 ? | ? | ?
---+---+---
 ? | ? | ?
```

You will replace ? with the appropriate display symbol for each index in the board. For example, if `board` is equivalent to

```
[[UNSEEN_WATER, UNSEEN_WATER, SEEN_SHIP],
 [SEEN_WATER, UNSEEN_WATER, SEEN_WATER],
 [UNSEEN_WATER, UNSEEN_WATER, UNSEEN_WATER]]
```

Then `display_board(board)` will print

```
   |   | x
---+---+---
 o |   | o
---+---+---
   |   |
```

Hint: Which function that you wrote can you use in this function?

Related Reading: 4.3, 5.2, 5.3, 6.2, 6.4

4 Arranging Ships

In this task, you will be writing the general battleship ship placement functions in `battleships.py`.

4.1 ship_at_pos()

This function takes a 2D list `board` and tuple `pos` and returns a boolean value of whether a ship is at `board[pos[0]][pos[1]]`. You may assume that `pos` will *not* be a board position that has been seen by an opponent.

Related Reading: 3.2, 6.2, 6.4

4.2 place_ship()

This function takes the following parameters

- `board`: A 2D list.
- `ships`: A list of the lengths of each ship. `ships[i]` gives the length of the ship with ID `i`.
- `ship_id`: The ID of the ship to place.
- `pos`: The position to place the ship.
- `direc`: The direction to place the ship in.

You may assume that `ship_id` is a valid index in `ships`. You should only place `ships[ship_id]` onto `board` if it can be placed onto the board without extending beyond the board or being put on top of another ship that has already been placed. If the ship was placed successfully, return `True`. If not, return `False`. For example, suppose you have `board`

```
[[UNSEEN_WATER, UNSEEN_WATER, UNSEEN_WATER],
 [UNSEEN_WATER, UNSEEN_WATER, UNSEEN_WATER],
 [UNSEEN_WATER, UNSEEN_WATER, UNSEEN_WATER]]
```

`ships = [3,2]`, `ship_id = 1`, `pos = (0,0)`, and `direc = 'e'`. This ship can be placed successfully and should result in `board` being

```
[[1, 1, UNSEEN_WATER],
 [UNSEEN_WATER, UNSEEN_WATER, UNSEEN_WATER],
 [UNSEEN_WATER, UNSEEN_WATER, UNSEEN_WATER]]
```

and `place_ship` returning `True`. Note here, as discussed in the Game Overview, that we put the ID of the ship on the board where it is placed so that when hits are made on the ship, we can update the amount of life a player's ships have left.

Hints: Which function that you wrote can you use in this function? You can only place a ship successfully with certain constraints. Which constraints can you check directly and return early? Which constraints require more complex checks?

Related Reading: 3.1-3.3, 3.5, 4.2, 6.2, 8.4, 8.5

4.3 filtered_print()

This function takes a list of integers `ls` and a list of booleans `filter` and prints the elements `ls[i]` where `filter[i]` is `True`. An empty string "" is printed for `ls[i]` where `filter[i]` is `False`. For example, if `ls=[5,9,0]` and `filter=[True,False,False]` then `filtered_print(ls,filter)` prints

```
[5,,]
```

You may assume that `ls` and `filter` have the same length.

Related Reading: 3.5, 4.2, 5.1-5.3, 6.2

4.4 place_ships()

This function takes a 2D list `board` and list `ships` and places each ship onto the `board`. Your algorithm should do the following while there are still ships left to place.

- Print which ships (in particular, their lengths) still need to be positioned.
- Prompt the user to position a ship. They will specify a ship ID, row, column, and direction separated by spaces. For example, to place ship 0 facing north at the 1st row 2nd column the user would enter `0 1 2 n`.
- If the ship ID is valid and the corresponding ship hasn't been placed already, the ship is attempted to be placed.

For example, suppose you have a board

```
[[UNSEEN_WATER, UNSEEN_WATER, UNSEEN_WATER],
 [UNSEEN_WATER, UNSEEN_WATER, UNSEEN_WATER],
 [UNSEEN_WATER, UNSEEN_WATER, UNSEEN_WATER]]
```

and `ships = [3,2]`. Suppose we call `place_ships(board, ships)`. The console interaction could look like:

```
[3,2]
Position? 0 0 0 e
[,2]
Position? 1 1 1 w
```

Here, the lengths of the ships that need to be positioned are printed first. Then, the user positions the 0th ship at (0,0) facing east. Now, only the 2 length ship is printed. Then, the user positions the 1st ship at (1,1) facing west. You can prompt and convey information to the user in whatever way you choose, but you **need** to include the lengths of the ships left to position before each iteration. `board` should end up as

```
[[0, 0, 0],
 [1, 1, UNSEEN_WATER],
 [UNSEEN_WATER, UNSEEN_WATER, UNSEEN_WATER]]
```

You can assume that the user will enter ship ID (integer), row (integer), column (integer), and direction (string) separated by spaces. But remember, you will have to check the validity of the ship ID and whether a placement can be made.

Hints: This function is quite challenging. Here are some ideas on how to implement it.

1. Notice in the example how the lengths of the ships that are not positioned are printed. Which function will print output matching this manner?
2. Based on your answer to hint 1, what other variables will you have to maintain which ships have been positioned?
3. Which function can you use to try to place a ship? Based on your answer, does the function tell you whether placement was successful?
4. Based on your answer to hint 3, you may need to parse the user input into a collection of variables. Which function can you use to do this?

Related Reading: 1.3, 3.3, 3.5, 4.1, 6.1-6.3, 8.6, 9.1, 9.2

5 Player Turns

In this task, you will be writing the general battleship turn execution and processing functions in `battleships.py`.

5.1 next_player()

This function does not take any input or produce any output. It will be used when switching between turns when playing the battleship game. All it should do is prompt the user to enter something and press enter.

Hint: It should only be 1 line of code.

Related reading: 9.1

5.2 player_turn()

This function should implement interaction with a player during their turn. It takes an integer board size (`size`) as an input parameter. As noted in the Introduction, players will take shots at their opponents ships. They will be entering a row and column to shoot at separated by a space. This function should keep asking the user to enter a valid row and column. You can assume that the user will always input two integers separated by a space. Once a valid row and column are input, return them as a tuple.

Hint: Why do you think the `size` parameter is provided?

Related Reading: 1.3, 3.3, 3.5, 4.1, 7.2, 9.1, 9.2

5.3 process_turn()

This function takes a 2D list `board`, list of `ships`, and a tuple `pos`. `pos` will be the position a player fired at in `player_turn()` (so you can assume it's on the `board`). This function should do the following:

- If firing at `pos` hits a ship, you should lower the ship's health in `ships`. You should also update `board` to note that the player has seen a ship here. If this sunk the ship, return `SUNK_ID`, otherwise return `HIT_ID`.
- If firing at `pos` hit water, update `board` to note that the player has seen water there. Then, return `MISS_ID`.

Hints: This function is also somewhat complicated.

1. How will you identify which ship in `ships` should have its health lowered? What are we storing in `board` at the start of the game?
2. How do we determine if this hit sunk the ship as opposed to just hitting it?
3. How will we be updating `board` in this function when we see water or a ship?

Related Reading: 3.3, 3.5, 3.6, 6.2, 6.4, 8.4-8.6

6 Imports in two_player_battleships.py

In order to implement `two_player_battleships.py`, which file(s) do you need to import? Feel free to use aliases.

Hints: Remember, you will be writing functions that are specific to a two player game, but will have a variety of existing battleships functions to start with. Also, one of the primary functions you will be implementing is `play_game()` which will need to take into account turn results.

7 Preparing a 2 Player Game

In this task, you will be writing the 2 player battleship set up functions in `two_player_battleships.py`.

7.1 init_boards_and_ships()

This function takes a list of `ships` that are available for a 2 player game and a board size `size`. You may assume `size > 0`. You will need to return four values

- `player1_board`: a 2D list representing the board of player 1 with size `size`.
- `player2_board`: a 2D list representing the board of player 2 with size `size`.
- `player1_ships`: a list of the lengths of the ships player 1 can choose from to place on their board. This will also be the starting amount of life for each of player 1's ships at the beginning of the game being played.
- `player2_ships`: a list of the lengths of the ships player 2 can choose from to place on their board. This will also be the starting amount of life for each of player 2's ships at the beginning of the game being played.

For example, if we had the following code

```
ships = [2,3]
p1_board, p2_board, p1_ships, p2_ships = init_boards_and_ships(ships, 3)
```

Printing `player1_ships` and `player2_ships` should output

```
[2,3]
[2,3]
```

Hints: How will you ensure that `player1_ships` and `player2_ships` can be updated separately during the game? You have written all of the necessary functions already, this function should be only 4 lines and a return statement.

Related Reading: 7.3, 8.6, 8.7

7.2 prepare_boards()

This function takes 4 inputs - boards for players 1 and 2 and lists of ships for players 1 and 2 - and prompts first player 1 then player 2 to place all their ships onto their boards. After player 1 is done positioning ships you should prompt the user to confirm that there will be a switch for player 2 to begin placing ships. This prompt can be anything as long as the user has to enter something.

Hint: You have functions that implemented all three of these steps. This function should only be 3 lines of code.

Related Reading: 8.5, 8.6

8 Playing 2 Player Game

In this task, you will be writing the battleship game execution functions in `two_player_battleships.py`.

8.1 game_over()

This function takes a list of ships for player 1 and a list of ships for player 2 and returns `True` if the game is over and `False` otherwise.

Hint: When is a 2 player game over? Or, as noted in the Introduction, when has one of the players won?

Related Reading: 2.2, 3.3, 4.2, 6.2, 8.3, 8.4

8.2 play_game()

This function takes 4 inputs, lists of ships for player 1 and player 2 and boards for player 1 and player 2 with their ships placed on their boards. This function should run a 2 player game and returns 1 if player 1 won and 2 if player 2 won. The game should start on player 1 and until the game is over

1. Display the current player's record of hits and misses.
2. Have the player select a position to target.
3. Process the player's position.
4. If it was a miss, go to the next player. Moving to the next player should prompt the user for confirmation with any prompt.

You are allowed to print additional information in this function, but you must print the current player's record at the beginning of each iteration until the game is over.

Hint: This function is somewhat complex because you need to fully understand all the functions you have written and how to use them.

1. The primary challenge of this function is managing who is the current player as well the board and ships they will be targeting.
2. Steps 1-3 can be implemented in one line *each* as you have functions that already implement these operations. You will need to choose what you pass into these functions (carefully).
3. You also have a function to implement part of step 4 and a function to check if the game is over.
4. If you don't display anything to the user from this function besides their record and use ternary operators extensively, this function can be written in 10 lines of code.

Related Reading: 3.6, 4.1, 8.6

9 Main script

In this task, you will be writing the main script that runs the game with any board size and collection of ships you want.

Hint: You have implemented all of the necessary functions to initialize, prepare, and play a 2 player game. This should only be 3 lines of code.

Related Reading: 8.6, 10.1