Project: Battleship

Battleship Rules

The board game “Battleship” is one of the earliest games to have a computer version. Battleship is a game where players place rectangular ships on a grid (the “battleships” in question), and try to guess where their opponent’s ships are by firing “missiles” onto squares of an ***n* × *n*** grid in order to sink their opponent’s battleships.

The first person to sink all the enemy battleships wins. Each player has 5 ships which they can choose to place in a horizontal or vertical orientation.

The game is played in two phases:

1. Players secretly place their ships on a grid such that no two of a particular player’s ships overlap or are diagonal.
2. Players take turns guessing single **(*x*, *y*)** coordinate pairs where the opponent’s ship could be. The opponent must declare whether this coordinate resulted in a “hit” (one of their ships was damaged) or a “miss” (the coordinate is “in the ocean”).

- If a player gets a “hit”, they can keep on guessing until they “miss”.
- If a player gets a “miss”, their turn is over, and the other player can start guessing.

In this project, you will work on your own version of a two-player Battleship computer game.

Battleship Helpers

In the first part of the project, we will implement some of the key logic behind the implementation of Battleship.

Switching players

First, we need a way of switching players. In our code, player 1 is represented as **0** and player 2 is represented as **1**. Your first task is to implement a function that, given a player number, **return**s the opponent’s player number.

get_opponent(player)

This function finds the opponent of the current player. It should return 0 if **player** = 1 and 1 if **player** = 0. The behavior of this function when given an argument that is not **0** or **1** is undefined but the program may not crash.

Task 1. Implement the **get_opponent** function in **task1.py**.

Input processing

In our implementation of Battleship, players use a **7 × 7** grid to position their ships, with coordinates ranging from 1 to 7 on both dimensions. Each player is required to place:

- 1 ship of size 4,
- 2 ships of size 3, and
- 2 ships of size 2

To place their ships, players specify the grid coordinates, **(*x*, *y*)**, for the upper-left corner of the ship, and then choose whether to rotate the ship from its default horizontal orientation to a vertical orientation using terminal input (Y or N).

The **7 × 7** board is represented by a 2D list where **x** is the column index and **y** is the row index. The upper-left-hand corner of the grid is location **(0,0)**.

In the next few tasks, we will focus on processing player coordinate inputs. Specifically, you will need to complete the implementation of the following functions:

check_valid_coord(coord, board)

This function checks if the integer coordinates are valid on the given board, returning **True** if they are within the grid boundaries and **False** otherwise. For extra credit, you may not hardcode the board size as 7; the function should be adaptable to any board size!

Task 2. Implement the **check_valid_coord(coord, board)** function in the **task2.py** file.

ask_coordinates(board)

This function repeatedly requests user input for **(x, y)** coordinates until they enter a valid input. An input is considered valid if the following are all true:

- The input string contains a comma “**,**”.
- The substrings on either side of the comma must represent integers.
- The coordinates must be within the board boundaries and the board cell at those coordinates must be **False**.

You should use the **.split()** and **.isdigit()** methods when writing this function.

Task 3. In the **ask_coordinates(board)** function in the **task3_5.py** file, write the part of the code that splits the string input from the player by comma.

Task 4. In the **ask_coordinates(board)** function in the **task3_5.py** file, write the part of the code that checks that the user input is valid by checking that the input represents two digits separated by a comma.

Task 5. In the **ask_coordinates(board)** function in the **task3_5.py** file, write the part of the code that casts the coordinates to integers, and modifies their values to make them suitable for indexing. The x-coordinate should be saved as **x** and the y-coordinate as **y**.

Initializing the boards

Our implementation of Battleship relies on 4 different **boolean** boards: ship and hit boards (one of each, for each player). On the ship boards, **True** indicates that a player has placed part of a ship at that location (and **False** indicates no ship). On the hit boards, **True** indicates that a player has attacked that location on their opponent’s ship board. In this part of the project, we will implement the function that initializes a board to all **False** values:

make_boolean_square_board(size)

This function creates a square 2D board of the specified size, initializing each coordinate cell with the value **False**. It returns a list of lists representing the board with each sublist representing a row.

Task 6. Implement the **make_boolean_square_board(size)** function in the **task6.py** file.

Ship Placement

In the second part of the project, we will complete the key logic behind placing the ships before the game starts. We will work in the **battleship.py** file for this part.

As an overview, the function we will be completing is called **place _ ships** and has the following specification:

place_ships(player, rounds, board_ships_p, ships_placed)

This function allows the player to place their ships on the board. The following actions take place:

- Visualize the current state of the board with existing ships.
- For each ship, prompt the player to place the ship on the board. Validate the ship's position and handle its orientation based on player input.
- Ensure the ship fits within the board boundaries and does not overlap with existing ships. If the placement is invalid, prompt the player to try again.
- Update the board and record the ship's coordinates in the **ships_placed** list.
- Clear the screen to display the updated board with the current player and round information.

To finish writing this function, we'll break the task into smaller parts and implement our plan, piece by piece.

Task 7. Get the size of the ship currently being placed and call it **l**. Then, **print** the size of the ship to the user in this format: "Place a ship of size **l**!"

Task 8. Append the coordinates covered by the ship to the **new_ship** list, representing the ship's position when positioned vertically.

Task 9. Repeat the last task for when the ship is positioned horizontally.

Task 10. Finally, add **new_ship** to **ships_placed** (which is a list of coordinate tuples for every ship).

Game Loop

In this section of the project, we will complete the implementation of the **attack** function, which is called when a player “attacks” their opponent. The function works as follows:

attack(player, rounds, board_ships, board_hits,ships_placed)

This function returns a boolean indicating whether the attack was a miss (**True** if so). The following steps and checks are performed during the attack:

- The current player is prompted to choose coordinates to attack on their opponent’s board.
- The selected coordinates are marked as **True** on the player’s hit board.
- If the attack hits an opponent’s ship, the function checks if all of the opponent’s ships have been sunk. This is done by verifying that no “unhit” ship parts remain on the opponent’s board.
- If all ships are sunk, the current player is declared the winner!
- If the attack misses, the function returns **True**, ending the player’s turn. Otherwise (if it returns **False** meaning a hit), the player gets another turn.

In this function, we will complete the step in which we check that all of the opponent’s ships have been sunk, comparing the player’s hit board to the opponent’s ship board.

Task 11. Finish the implementation of **check_win** in **task11.py**. Check whether for every ship position on the opponent’s ship board, a hit exists on the player’s hit board. If that is not the case, **won** should be set to **False** as there remains “unhit” ship parts on the opponent’s board.

Running the game!

As part of the **game_loop**, the round number must increase once both players have finished their turn in the current round.

Task 12. Implement **go_to_next_round** function in the **task12.py** which is called at the end of each player’s turn.

Task 13. It's time to actually play the game! Open up **battleship.py** and click on the play button in the top right to run the game. You might need to make your terminal bigger to fit all the content, but you should see the game instructions and play the game with a friend.