BATTLESHIP

15-110 S17: Programming Assignment 9

Due May 2, 2017 @ 10pm — Checkpoint due April 14, 2017 @ 10pm

Battleship Overview

Battleship is a two player game in which players place ships on a grid and take turns guessing at the location of their opponent's ships. The goal is to be the first player to guess all the locations of each of the opponent's ships, or to sink each of them. This assignment's implementation of Battleship has the following rules:

- The game has two 10 x 10 grids, or boards, for each player. The individual squares on the grid are identified by letters (down the left side of the board, corresponding to each row of the grid, A through J), and numbers (across the top of the board, corresponding to each column of the grid, 0 through 9).
- For each player, one of the boards, called the play board, manages the locations of the player's ships and records the hits they receive from the opposing player. The second board, called the guess board, only records the player's hits and misses (guesses) on the opponent's ships. Each player cannot see their opponent's arrangement of ships.
- Ships are represented as consecutive squares on a board, arranged either horizontally or vertically. Ships cannot overlap on a given board.
- Each player has 5 ships, each of which has a different fixed size, meaning the number of squares occupied. The 5 ships are: Carrier (size 5), Battleship (size 4), Cruiser (size 3), Submarine (size 3), and Destroyer (size 2).
- Before the game starts, each player places their 5 ships on their play board without the opponent's knowledge of where they are placing them.
- In each turn, the current player picks a target space on the opponent's board to try and strike one of their ships. The player receives feedback as to whether their guess was a hit or a miss.
- If the player successfully hits one of the opponent's ships, then in the following turns, the relevant square is colored red on both the player's guess board and the opponent's play board.
- Otherwise if the target was a miss, the square will be marked blue only on the player's guess board, and nothing will change on the opponent's play board.
- When all squares of a player's ship have been hit, the ship is sunk. When all the ships of a given player are sunk, the game ends and the opponent wins the game.

Important Notes

Please note that the starter file for the assignment is available on Autolab. Make sure to read the entire write-up before you begin working on this assignment. Pay special attention to the **Testing Your Code** and **Style** sections at the end of the writeup. We have guidelines for testing your code and style for this assignment that differ from previous assignments. Keep in mind the that the checkpoint is **less** than half way through the assignment and that you should start early on the View and Controller portions of the assignment.

Checkpoint

The due date for the checkpoint is April 14. For the checkpoint, you should submit the Model portion of the assignment (1.1 through 1.10). The functions for the checkpoint **will** be autograded.

1. Model (7 points)

Overview

The Model handles the Python representation of the Battleship game. Our implementation of the model includes:

- Each board (guess boards and play boards for each player 1 and player 2) is a 2-dimensional list where each element in the list represents a square on the Battleship grid. The contents of the board are as follows:
  - ▷ **0** represents a square that is essentially empty, meaning it has no ship on the play board, and no hit or miss on the guess board.
  - ▷ **1** represents a square that is occupied by a ship, but has not been hit.
  - ▷ **2** represents a square that has been hit (on the current player's play board this means one of the current player's ships has been hit at the given location, on the current player's guess board this means the current player hit one of the opponent's ships at this location).
  - ▷ **3** represents a square that is a miss (this can only occur on the guess board of each player).
- Individual ships are represented as lists of tuples where the tuples are **row, column** coordinate pairs, for example (1, 4) represents the space B4 on the grid. The ship will then be a list of all of the coordinates which it occupies.
- Each players ships are contained in a dictionary mapping each ship's name to it's list representation. For example, the key-value pair `'Destroyer': [(0,1),(0,2)]` might be an entry in the ship dictionary for a player. The dictionary will contain 5 key-value pairs, for 5 ships.
- The dictionary `SHIP_SIZES` is defined for you in the starter code. This maps the names of the 5 ships to their lengths.
- Players are represented by integers 1 and 2 for Player1 and Player2, respectively.

For reference, here is a table of each of the global variables that will be necessary to implement Battleship and their types:

Variable	Type
PLAYER1_GUESS_BOARD	2D list
PLAYER2_GUESS_BOARD	2D list
PLAYER1_PLAY_BOARD	2D list
PLAYER2_PLAY_BOARD	2D list
PLAYER1_SHIPS	dictionary
PLAYER2_SHIPS	dictionary
SHIP_SIZES	dictionary (already defined)

Model Tasks:

1.1 `init_board()`
Write the function `init_board` which returns the initial list that we will be using as our model to represent a 10 x 10 board. Notice that 0 at a location on the board means that there is no ship, hit, or miss at that location, and our initial board should have no ships.

1.2 `is_ship_on_board(ship)`
Write the function `is_ship_on_board` which takes in `ship` (the coordinate list of a ship, as represented in the way we described above), and returns True if all coordinates of the ship fall within the range of the 10 x 10 board and False otherwise.

1.3 `same_row(ship)` and `same_column(ship)`
Write the function `same_row(ship)` and the function `same_column(ship)` which both take in `ship` (the coordinate list of a ship, as represented in the way described above). The function `same_row(ship)` returns True if all squares the ship occupies are in the same row and False otherwise. Similarly, the function `same_column(ship)` returns True if all squares the ship occupies are in the same column and False otherwise. The coordinates list is not necessarily ordered (by row or by column).

1.4 `is_valid_ship(ship, size)`
Write the function `is_valid_ship` which takes in `ship` (the coordinate list of a ship), and `size` (the size this ship should be based on the type of ship the coordinate list corresponds to). The function should return True if **all** of the following criteria are met, and False otherwise. A ship is valid if:

- it is arranged either completely horizontally **or** completely vertically.
- the ship size corresponds to the number of coordinates it takes up.
- all ship coordinates are on the board.
- the ship occupies consecutive squares on the board.
    - Hint: consider sorting the coordinates and checking that either the rows (if the ship is vertical) or columns (if the ship is horizontal) are in ascending order.

You should use `is_ship_on_board`, `same_row`, and `same_column` that you defined above to check these conditions.

1.5 `is_valid_placement(ship, board)`
Write the function `is_valid_placement` which takes in `ship` (the coordinate list of a ship), and `board` (the board onto which the ship is to be placed), and returns True if all of the squares that the ship tries to occupy on the board are empty and False otherwise.

1.6 `place_ship(ship, board)`
Write the function `place_ship` which takes in `ship` (the coordinate list of a ship), and `board` (a play board), and modifies `board` so that the relevant coordinates show where the ship is. This function should return None. You may assume the ship is valid for placement on the board.

1.7 `mark_hit(board, row, col)` and `mark_miss(board, row, col)`
Write the function `mark_hit` and the function `mark_miss` which take in a board, `board`, `row`, and `col`. In `mark_hit`, mark the square at the position specified by `row` and `col` to be a hit. In `mark_miss`, mark the square to be a miss. You may assume `row` and `col` are valid coordinates for the board. Both functions should return None. Note: `mark_hit` will be used on both guess boards and play boards, while `mark_miss` is only used on guess boards.

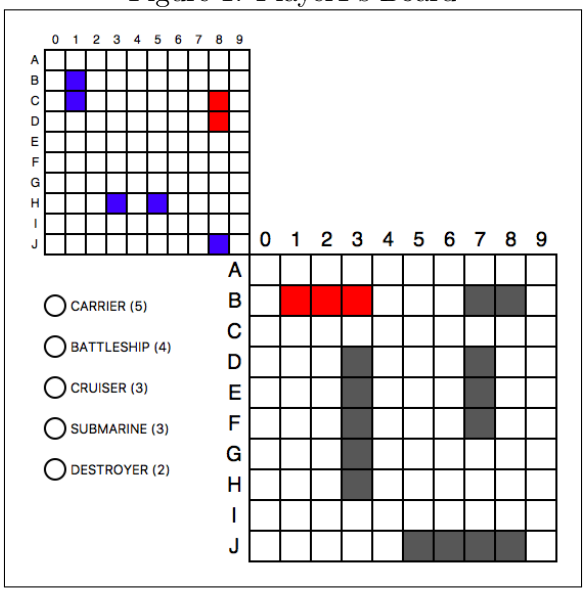
1.8 `is_hit(board, row, col)` and `is_miss(board, row, col)`
Write the functions `is_hit` and `is_miss`, which take in a `board` (one of the players' play boards), `row`, and `col`. These functions determine the status of an attempted strike at the coordinates (`row`,`col`). In `is_hit`, return True if the square specified by `row` and `col` is filled by a ship and False otherwise. In `is_miss`, return True if the square is marked as empty and False otherwise. You may assume `row` and `col` are valid coordinates for the board.

1.9 `remove_location(opponent_dict, opponent, row, col)`
In this implementation, when a guess is a hit to one of the opponent's ships, we remove that coordinate from the ship list of the corresponding ship in the opponent's dictionary of ships. This serves to keep track of how many spaces of a ship remain to be hit. When a ship list is emptied of all of its coordinates, it has been sunk.

Write the function `remove_location` which takes in `opponent_dict` (the opponent's dictionary), `opponent` (the number of the current player's opponent), and `row` and `col`. The function should remove the coordinate `(row, col)` from the opponent's corresponding ship's coordinate list in `opponent_dict`. If this coordinate is the last remaining coordinate occupied by the ship, that ship has been sunk. In this case, print a message 'YOU SUNK PLAYER{*opponent's player number*}'S {*ship name*}'. If the coordinate is not anywhere in the dictionary, print 'ERROR: COORDINATE NOT FOUND'. The function should return None.

1.10 `all_ships_sunk(ship_dict)`
Write the function `all_ships_sunk` that takes in one of the player's ship dictionary, `ship_dict`, and checks whether or not all of the ships in the dictionary are sunk. It should return True if all the ships are sunk, and False otherwise. Recall that if a ship is sunk, then the coordinate list of the ship will be an empty list.

Figure 1: Player1's Board



2. View (6 points)

Overview

You have a considerable amount of freedom in how you design the display of your Battleship game. We require that you display both the play board and guess board of the current player along with their row and column labels. We also require that your display includes something that marks the opponent's ships as they are sunk. In addition, we require splash screens to be displayed between turns that display whether the turn just taken resulted in a hit or miss, and prompts the next player to begin their turn. A view that fulfills these requirements in a way that is readable and clear to the user will receive full credit. **Note that while there is opportunity to spend a lot of time on the view portion of this assignment we recommend you implement a basic view, finish the assignment, and then come back to make improvements to the view**.

We have provided for you an example of a view that implements all of these things. Figures 1 above and Figure 2 below are the boards shown to Player1 and Player2, respectively. The upper left corner is the player's guess board and the lower right corner is the player's play board. Grey corresponds to a ship, red corresponds to hits, and blue corresponds to misses. Notice how Figure 2 has filled in red that Player1's Cruiser has been sunk (shown on the guess board directly above). The ship list displays those ships belonging to the **opponent** that the current player has successfully sunk. Tables 1 and 2 show what the key-value pairs in each player's ship dictionary look like at the start of the game, given their placements on the board. (Note that the figures show some of the coordinates as being hit, meaning that some of the coordinates in the ship lists in the table will have been removed in each player's ship dictionary.)

View Tasks:

2.1 `draw_guess_board(board, start_x, start_y)`
Write the function `draw_guess_board` which takes in `board` (the current player's guess board), and `start_x` and `start_y`, the x-coordinate and y-coordinate of the upper left corner of the board. The function draws the given board. For a specific square, mark it red if there is a hit on the square, and blue if there is a miss. Empty squares should be colored white. The function **does not** need to draw the coordinates along the left and top edges.

Table 1: Player1's Ships

Variable	Type
'CARRIER'	[(3,3),(4,3),(5,3),(6,3),(7,3)]
'BATTLESHIP'	[(9,5),(9,6),(9,7),(9,8)]
'CRUISER'	[(1,1),(1,2),(1,3)]
'SUBMARINE'	[(3,7),(4,7),(5,7)]
'DESTROYER'	[(1,7),(1,8)]

Figure 2: Player2's Board

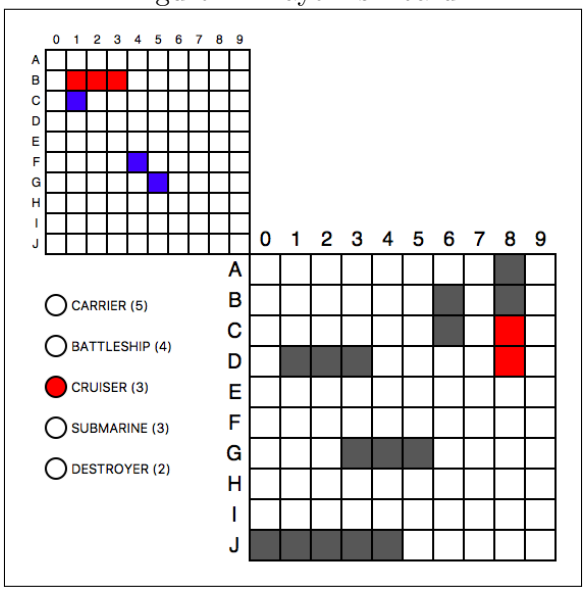


Table 2: Player2's Ships

Variable	Type
'CARRIER'	[(9,0),(9,1),(9,2),(9,3),(9,4)]
'BATTLESHIP'	[(0,8),(1,8),(2,8),(3,8)]
'CRUISER'	[(3,1),(3,2),(3,3)]
'SUBMARINE'	[(6,3),(6,4),(6,5)]
'DESTROYER'	[(1,6),(2,6)]

2.2 `draw_play_board(board, start_x, start_y)`
Write the function `draw_play_board` which takes in `board` (the current player's play board), and `start_x` and `start_y`, the x-coordinate and y-coordinate of the upper left corner of the board. The function draws the play board of the input player. For a specific square, mark it red if it is hit by opponent and gray if it is occupied by a ship. Empty squares should be colored white. The function **does not** need to draw the coordinates along the left and top edges.

2.3 `draw_coords(start_x, start_y, box_width, font_size)`
Write the function `draw_coords` which takes in `start_x` and `start_y`, the x-coordinate and y-coordinate of the upper left corner of the board they are displaying the coordinates for, `box_width`, the length of the side of a single box of the board, and `font_size`, the font size of the text. The function should draw the letters (A-J) down the left side of the board, aligned with the rows of the board, and numbers (0-9) across the top of the board, aligned with the column. You will need to do some preliminary adjustment of `start_x` and `start_y` to position the letters and numbers so that they are drawn alongside the board and do not overlap the board.

2.4 `draw_ships(opponent_dict, start_x, start_y)`
Write the function `draw_ships` which takes in `opponent_dict` (the opponent's ship dictionary) and `start_x` and `start_y`, the x-coordinate and y-coordinate of the upper left corner of the ship list. The function draws an overview of the state of the opponent's ships. The function draws the names of the ships next to circles. A circle is filled in with red when the corresponding ship has been sunk. You should also display the original size of each ship next to its name.

2.5 `draw_splash_screen(player, hit)`
Write the function `draw_splash_screen` which takes in `player` (the integer representing the player whose turn is next), and `hit`, a boolean value indicating whether the previous turn resulted in a hit or miss (True if hit, False if miss). The function should draw a screen that overwrites everything on the canvas, and draw a message showing whether the past move was a hit or miss, and draw a message indicating the next player. See Figure 3 for reference.

2.6 `display_board(player)`
Write the function `display_board` which takes in `player`. The function should draw the input player's guess board, play board, and listing of opponent's ships by calling `draw_guess_board`, `draw_play_board`, and `draw_ships`. Each function should be called with arguments for the current player and your desired `start_x` and `start_y` positions for each item. The function should also draw the coordinates along the edges of both the guess board and the play board by calling `draw_coords` twice with the relevant arguments. At the beginning of the function, it deletes everything on the canvas, and at the end it updates the window. These lines have been implemented for you.

The first three lines given in the function are calls to functions you will write in the controller section that retrieve the relevant global variables given a player. They are used to define variables `guess_board`, `play_board`, and `opponent_dict`. You should not modify these lines and you should use these variables as arguments to your drawing functions. However, if you wish to test the function before you've defined the controller functions you can temporarily replace the function calls with other boards/dictionaries.

Figure 3: Splash Screen Between Turns

3. Controller (7 points)

Overview

The Controller is responsible for taking user input from the terminal, updating the model accordingly, and calling the view functions to display the current state of the game to the users. The controller will implement:

- Prompting each player to input locations in which to place their ships.
- Verifying that the given input is valid and transferring the input to a representation consistent with the model.
- Processing each player's turn:
  - Collecting input for a target box on the board.
  - Checking the validity of the input, then updating the model accordingly.
  - Displaying the outcomes of each turn.
- Switching between players.
- Checking if the game has finished.

Controller Tasks:

3.1 `init_game()`
Write the function `init_game` that takes no arguments and initializes the play board, guess board, and ship dictionary for both players. Note that all four boards and both ship dictionaries have been initialized at the top of the function as global variables. This is so they may be modified by other functions without needing to be passed in as arguments or returned from each function. Do not modify these lines of code. These are variable names you should use throughout your program when referring to these boards and dictionaries. In this function you should initialize each board using `init_board` and initialize both dictionaries to be empty dictionaries.

3.2 `get_player_ships(player)`
Write the function `get_player_ships` which takes in an integer representing a player and returns this player's ship dictionary.

3.3 `get_player_guess_board(player)`
Write the function `get_player_guess_board` which takes in an integer representing a player and returns this player's guess board.

3.4 `get_player_play_board(player)`
Write the function `get_player_play_board` which takes in an integer representing a player and returns this player's play board.

3.5 `get_opponent(player)`
Write the function `get_opponent` which takes in an integer representing the current player and returns an integer representing the player's opponent.

3.6 `is_valid_ship_input(input_list)`
Write the function `is_valid_ship_input` which takes in `input_list`, a list of strings, where each string identifies a square on the board. The function returns True if the input is of the valid form. Notice that each string should have two characters: The first one should be a letter (representing row number), and the second is a number (representing column number).

The function should check that:

- The length of each string in the input list is 2.
- The first element is a capital letter 'A' through 'J' inclusive.
- The second element is a number 0 through 9 inclusive.
- The input list is not empty.

The function **does not** check that the pairs are next to each other; that is handled in the model function `is_valid_ship`.

Example usage:

```
>>>is_valid_ship_input([‘A1’,‘A2’,‘A3’])
True
>>>is_valid_ship_input([‘A1’,‘B3’,‘C4’])
True
>>>is_valid_ship_input([‘A1’,‘K3’,‘C4’])
False
>>>is_valid_ship_input([‘AA1’,‘B’,‘C4’])
False
```

3.7 `convert_input(input_list)`
Write the function `convert_input` which takes in `input_list`, a list of strings, where each string identifies a square on a board by the *letter-number* string form as described in the previous question. The function converts each string of the list to a tuple (row, col) that represents the actual row number and column number of the square on the board. The function should return the resulted

new list of tuples that represents a ship.

Example usage:
```
>>>convert_input(['A0','B3','E4'])
[(0,0),(1,3),(4,4)]
```

3.8 `pick_ships(player, board, ship_dict)`
The function `pick_ships` is responsible for prompting a player to place each of his/her ships on their play board, checking the validity of the given input, and placing each ship on the player's play board.

Write the function `pick_ships`, which takes `player`, an integer representing a player, `board`, the player's board, and `ship_dict`, the player's ship dictionary. The function should loop over each of the ships in the global `SHIP_SIZES` and prompts the player to give input of the appropriate size. Each square is represented using the *letter-number* string form described in previous questions. For example, an input might look like `'A0 A1 A2'`. (Note that the input function returns the user input in the form of a string and that the user will input coordinates without the quotations.) The while loop given in the starter code loops as long as the player has not given a valid input, using `is_valid_ship_input`, `is_valid_ship` and `is_valid_placement` and continues prompting the user until their input is valid. This loop is implemented for you. Once valid input is obtained for each ship you must:

- Convert the input using `convert_input` into a ship list.
- Place the ship on the `board`.
- Insert the ship into the player's ship dictionary, `ship_dict`, with the ship name as the key and the ship list as the value.
- Display the board for the given player so they can see their newly placed ship.

3.9 `is_valid_move_input(move)`
Write the function `is_valid_move_input` which takes in `move`, a string representing a square on the board. The function returns True if the string follows the *letter-number* way of identifying a valid square on the board.

A move is valid if:

- Its length is 2.
- The first element is a capital letter 'A' through 'J' inclusive.
- The second element is a number 0 through 9 inclusive.

3.10 `is_valid_move(board, row, col)`
Write the function `is_valid_move` which takes in `board`, the current player's guess board, `row`, a row number, and `col`, a column number. The function returns True if the square specified by the `row` and `col` on `board` has not been hit or missed before.

3.11 `is_end_game()`
Write the function `is_end_game` which takes in no arguments and which returns True if either player has all of their ships sunk and False otherwise. If either player has all their ships sunk print a message

to indicate that the game is over and who the winner is. You should call the function `all_ships_sunk`.

3.12 `play_game()`

Write the function `play_game` which is responsible for running the game play loop for the entire game.

Before the loop begins, the function should:

- Initialize the game by calling `init_game`.
- Display the board for Player1 and call `pick_ships` for Player1.
- Display the board for Player2 and call `pick_ships` for Player2.
- Initialize the variable `player` to `1`.

The game play loop is responsible for alternating turns until the game has finished. In each turn the function should:

- Prompt the current player to input a coordinate of a square on the opponent's board to strike. (Like in `place_ships`, the loop that continually prompts the user until a valid input is given is implemented for you).
- Convert the input into a row integer and a column integer.
- Determine if the given row and column are a hit or a miss on the opponent's play board using `is_hit` and `is_miss`.
  - If it's a hit, the function should mark the hit on the opponent's play board and on the current player's guess board using `mark_hit` and remove the hit coordinates from the opponent's ship dictionary by calling `remove_location`. It should also print 'HIT'.
  - If it's a miss, the function should only mark the miss on the current player's guess board using `mark_miss`, and print 'MISS'.
- Switch and update `player` at the end of the turn.
- Call `draw_splash_screen`, and immediately after, use the input function to prompt the new player to press enter to begin their turn.

When the loop has ended, the function should call `quit_game` to end the game.

Optional Bells and Whistles (up to 2 points extra credit)

You may choose to add some additional features to your Battleship game for extra credit. Suggestions of good bells and whistles include but are not limited to:

- implementing an AI for a player to play against.
- particularly striking graphics.
- adding sounds to the game.
- implementing mouse clicks to select targets on the guess board.

Testing Your Code

The model portion of the assignment will be autograded. For the view and controller portions **there is no autograder** that will run your code at the time of submission. You are responsible for testing the correctness of your code, both on a function by function basis, and by playing the game thoroughly to check for potential bugs. For the model portion of the assignment we have included a small number of tests in the file `test_pa9.py` to help clarify the intended functionality of each function. You should, however, add your own tests to this file in order to ensure the correctness of your code.

Helpful Notes for Making Testing Easier

- The best way to find bugs in your code is to play the game all the way through. You can do this by calling `play_game()` when running the file in the Python shell. Since playing the game requires first placing ships onto the board, and this can be time consuming when you're trying to test later parts of the game, we recommend first testing that the ship placement function works correctly. Then, once you've finished with that, modify the `SHIP_SIZES` to contain only one or two ships with small sizes (something like {'SHIP1' : 2}) so that you can shorten the ship placement process and more quickly get to testing the rest of the game. You can then change `SHIP_SIZES` back when you're ready to play the whole game.

Style (up to 2 point deduction)

For this assignment, you will be graded on code style. For full credit, you must adhere to the code style standards on the course resources page. For a big project like this one, it is essential to comment your code and indicate what you are trying to do.

Also, in your PA9 folder, please include a `README.txt` file. The README file should explain general things about the project. This does not exempt you from documenting your code in line with comments. This will not only help us grade your project, but it will also help you remember what exactly your code is doing and why. Also, please include how long you spent on the project and what you found the most challenging about the project.

Handing in Your Code

As usual, you should hand in your work on Autolab. We do not have a limit on the number of submissions you can make on Autolab for this assignment. We recommend that you hand in your work every time you make major changes. Please save all your work in a PA9 folder and compress the folder as a zip file before you submit your work. Your folder should only contain the following files:

- `pa9_battleship.py`
- `README.txt`

Once you've submitted, both the checkpoint **and** the final submission, make sure your code passes the autograder tests for the model. This will help ensure that your code is free of syntax or filename errors that may have come about as you made changes between the checkpoint and the final.