Contents

1 Basic Test Results

```
Starting tests...
Sat 01 Jun 2024 17:17:17 IDT
2dab2d5cdf944aa82b6efa9b26e8553f02dbe03f  -


Archive:  /tmp/bodek.51vlmx5k/intro2cs2/ex4/daniel.rez/presubmission/submission
  inflating: src/battleship.py


Running presubmit code tests...
6 passed tests out of 6 in test set named 'ex4'.
result_code    ex4    6    1
Done running presubmit code tests

Finished running the presubmit tests

Additional notes:

Remember to test your code.

Each line in the output of the test 'ex4_main' represents a call to a function in the helper file:

A tuple starting with 'S': A call to the function 'choose_ship_location'
A tuple starting with 'T': A call to the function 'choose_torpedo_target'
A list of boards: A call to the function 'print_board'

If there are extra calls, the test will fail saying there are no more inputs.

```

2 battleship.py

```python
################################################################
# FILE : battleship.py
# WRITER : daniel_riazanov , daniel.rez , 336119300
# EXERCISE : intro2cs ex4 2024
# DESCRIPTION: Implementation of battleships game (practising list of lists, loops, modular code, validations,
# module imports, clean code, etc.)
# STUDENTS I DISCUSSED THE EXERCISE WITH: None
# WEB PAGES I USED: None
# NOTES: None
################################################################

import helper


def init_board(rows=helper.NUM_ROWS, columns=helper.NUM_COLUMNS):
    """
    Initialize a game board with given dimensions, by default we will use constants from helper file

    Parameters:
    rows (int): Number of rows in the board.
    columns (int): Number of columns in the board.

    Returns:
    list: A 2D list representing the game board, with each cell initialized to the WATER constant.

    Raises:
    ValueError: If rows or columns are not positive integers.
    """
    # Input Validation: (nothing have been said about what we can suppose about input so for code stability
    if not isinstance(rows, int) or not isinstance(columns, int):
        raise ValueError("Rows and columns must be integers")
    if rows <= 0 or columns <= 0:
        raise ValueError("Rows and columns must be positive integers")

    # Creating the game board using list comprehension, filling each cell with the WATER constant
    initial_board = [[helper.WATER for _ in range(columns)] for _ in range(rows)]

    return initial_board


def cell_loc(name):
    """
    Convert a cell location from 'letterNumber' format to (row, column) tuple.

    Parameters:
    name (str): A string representing the cell location in the format 'letterNumbers' representing (column, row).

    Returns:
    tuple: A tuple (row, column) where row is the numeric value corresponding to the letter and
           column is the zero-based index.
            Returns False if the input format is invalid (for future validation modularity)


    """
    # Ensure the input length is at least 2 characters
    if len(name) < 2:
        return False

    # Split the input into column (letter) and row (number) parts. name[1:] and not name[1] because the nums can be > 9
    inp_column, inp_row = name[0], name[1:]

    # Checking if the first character is a single English letter
    # (since "we can suppose that there will be at max 26 columns")
    if not inp_column.isalpha() or len(inp_column) != 1:
        return False

    # Checking if the second character is integer
    if not inp_row.isdigit():
        return False

    # If passed, convert and return tuple
    column_index = letter_to_num(inp_column)
    row_index = int(inp_row) - 1

    # In coord representation first char represents row and second column
    return row_index, column_index


def letter_to_num(letter):
    """
    Convert a letter to its corresponding numeric value where A=0, B=1, etc.

    Parameters:
    letter (str): A single letter in upper or lower case

    Returns:
    int: The numeric value corresponding to the letter (A=0, B=1, etc.).
    """
    # "Small letter and num are valid args, but func must receive only big letter and num as valid input"
    letter = letter.upper()
    # "It is possible to suppose that input will be in the correct format 'letterInts', thus if starting value is A and
    # will be assigned 0, every following relative to A"
    letter_to_a_b_order = ord(letter) - ord('A')
    return letter_to_a_b_order


def valid_ship(board, size, loc):
    """
    Check if a submarine of the given size can be placed on the board at the specified location.

    Parameters:
    board (list of lists): The game board.
    size (int): The size of the submarine.
    loc (tuple): A tuple (index_row, index_column) representing the starting location on the board.

    Returns:
    bool: True if the submarine can be placed, False otherwise.

    Raises:
    ValueError: If the board is not a matrix (list of lists).
    """

    # Ensure all the arguments are valid so the func logic will return appropriate result
    if not all(isinstance(row, list) for row in board):
        raise ValueError("board must be represented as a matrix (list of lists)")
    if not helper.is_int(size):
        return False
    # Validating that the loc have passed validations, and we can place a ship here
    if not loc:
        return False

    row_index, column_index = loc

    # Checking starting position compared to board sizes:
    if row_index < 0 or row_index >= len(board) or column_index < 0 or column_index >= len(board[0]):
        return False

    # If starting position is valid, checking if len from starting position is valid
    # (since we're placing ship only vertically):
    if row_index + size > len(board):
        return False

    # If position is possible, checking whether the needed cells are empty
    # (since we're placing ship only vertically checking down on rows from the same column):
    for i in range(size):
        if board[row_index + i][column_index] != helper.WATER:
            return False

    # If got up here all tests passed
    return True


def create_player_board(rows=helper.NUM_ROWS, columns=helper.NUM_COLUMNS, ship_sizes=helper.SHIP_SIZES):
    """
    Creating the player's game board by initializing it with water cells and placing ships based on the provided sizes.

    Parameters:
    rows (int): Number of rows in the board. Defaults to the value defined in the helper module.
    columns (int): Number of columns in the board. Defaults to the value defined in the helper module.
    ship_sizes (tuple): Sizes of the ships to be placed on the board. Defaults to the value defined in the helper module.

    Returns:
    list of lists: A 2D list representing the player's game board after placing ships.
    """

    # Initializing the game board with water cells
    board = init_board(rows, columns)

    # Placing ships on the board based on the provided sizes from user
    for ship_size in ship_sizes:

        # Trying to get valid loc to place the ship
        helper.print_board(board)
        loc_as_string = helper.get_input(f"enter the top coordinate for the ship of size {ship_size}: ")
        loc_as_tupple = cell_loc(loc_as_string)

        # Repeat asking until the valid location is received
        while not (valid_ship(board, ship_size, loc_as_tupple)):
            print("not a valid location")
            helper.print_board(board)
            loc_as_string = helper.get_input(f"enter the top coordinate for the ship of size {ship_size}: ")
            loc_as_tupple = cell_loc(loc_as_string)  # row, column

        # Finally placing the ship and filling cells vertically based on ship size
        for i in range(ship_size):
            board[loc_as_tupple[0] + i][loc_as_tupple[1]] = helper.SHIP

    return board


def fire_torpedo(board, loc):
    """
    Update the game board based on the result of firing a torpedo at the specified location.

    Parameters:
    board (list of lists): The game board.
    loc (tuple): A tuple (index_row, index_column) representing the target location on the board.

    Returns:
    list of lists: The updated game board after firing the torpedo.
    """

    row_index, column_index = loc

    # Check if the location is within the bounds of the board (couldn't check in cell lock because didn't receive board
```

```python
    # dimensions to compare
    if row_index < 0 or row_index >= len(board) or column_index < 0 or column_index >= len(board[0]):
        return False

    # Check if the current cell is already damaged
    if board[row_index][column_index] in {helper.HIT_WATER, helper.HIT_SHIP}:
        return False

    # Update the cell on the board based on the target type
    if board[loc[0]][loc[1]] == helper.WATER:
        # If originally in cell was WATER, change to HIT_WATER
        board[loc[0]][loc[1]] = helper.HIT_WATER
    elif board[loc[0]][loc[1]] == helper.SHIP:
        # If originally in cell was SHIP, change to HIT_SHIP
        board[loc[0]][loc[1]] = helper.HIT_SHIP

    return board


def is_fleet_destroyed(board):
    """
    Check if the fleet on the given board is destroyed.

    Parameters:
    board (list of lists): The game board.

    Returns:
    bool: True if the fleet is destroyed, False otherwise.
    """
    for row in board:
        # If still there are ships on the board, player is not defeated
        if helper.SHIP in row:
            return False
    # Otherwise defeated
    return True


def create_computer_board(rows=helper.NUM_ROWS, columns=helper.NUM_COLUMNS, ship_sizes=helper.SHIP_SIZES):
    """
      Generate the computer's game board by randomly placing ships based on predefined sizes.

      Parameters:
      rows (int): Number of rows in the board. Defaults to the value defined in the helper module.
      columns (int): Number of columns in the board. Defaults to the value defined in the helper module.
      ship_sizes (tuple): Sizes of the ships to be placed on the board. Defaults to the defined in the helper module.

      Returns:
      list of lists: A 2D list representing the computer's game board after placing ships.
      """
    # Initialize the game board with water cells
    board = init_board(rows, columns)
    for ship_size in ship_sizes:
        # Find valid locations for the current ship size using the reusable function find_valid_locations
        valid_locations = find_valid_locations(board, ship_size)
        # Choose a random location from the valid locations using the helper function choose_ship_location
        loc = helper.choose_ship_location(board, ship_size, valid_locations)

        # Finally placing the ship and filling cells vertically based on ship size
        for i in range(ship_size):
            board[loc[0] + i][loc[1]] = helper.SHIP
    return board


def find_valid_locations(board, ship_size):
    """
     Find valid locations on the board where a ship of given size can be placed.

     Parameters:
```

```
     board (list of lists): The game board.
     ship_size (int): The size of the ship to be placed.

     Returns:
     set: A set of tuples representing valid locations on the board.
     """

    valid_locations = set()
    for row in range(len(board)):
        for col in range(len(board[0])):
            # Reuse valid_ship function to check if the current location can fit ship placement
            if valid_ship(board, ship_size, (row, col)):
                valid_locations.add((row, col))
    return valid_locations


def define_valid_targets(board):
    """
    Providing a set of valid targets from the board where a torpedo can be fired.

    Parameters:
    board (list of lists): The game board.

    Returns:
    set: A set of tuples representing valid target locations.
    """
    # Initializing an empty set to store valid target locations
    valid_targets = set()
    # Iterate through each cell on the board to check for valid targets
    for r in range(len(board)):
        for c in range(len(board[0])):
            # Check if the current cell contains WATER or SHIP, indicating a valid target (Accordingly to the
            # requirements, already damaged cell in not a valid target
            if board[r][c] in {helper.WATER, helper.SHIP}:
                valid_targets.add((r, c))

    return valid_targets


def print_boards(player_board, computer_board):
    """
    Print the game boards with a masked view of the computer's board, showing only the results of the player's turns.
    One flexible function handles the representation of the game.

    Parameters:
    player_board (list of lists): The player's game board.
    computer_board (list of lists): The computer's game board.

    Returns:
    None
    """
    # Represent players board as is (ships are visible)
    player_view = [[cell for cell in row] for row in player_board]

    # Run on the computer board and represent all HIT_WATER HIT_SHIP cells as they are, all other cells (including not
    # damaged ships) will be marked as WATER
    computer_view = [[cell if (cell in {helper.HIT_WATER, helper.HIT_SHIP}) else helper.WATER for cell in row] for row
                     in computer_board]

    # Print the game boards using the reusable function helper.print_board
    helper.print_board(player_view, computer_view)


def main():
    """
    The main function to run the battleship game.
    This function initializes the player and computer boards, prints the initial state of the game boards, and then
    iterates through circles, where each circle is player's and computer's turn. The course of the game is decided in
```

```python
    the end of each circle when one of the fleets is destroyed or the game ends in a tie. Once the game ends,
    the function asks the player if he wants to play again, and the game restarts or terminates accordingly.
    """
    while True:
        # Initialize player and computer boards
        player_board = create_player_board()
        computer_board = create_computer_board()
        # Print initial game boards
        print_boards(player_board, computer_board)

        while True:
            # Player's turn
            while True:
                # Get the target from the player
                input_to_hit = helper.get_input("Choose target: ")
                player_target = cell_loc(input_to_hit)

                # Keep asking until chosen target passes validations of player_target and fire_torpedo funcs
                if not player_target or not fire_torpedo(computer_board, player_target):
                    print('invalid target')
                    continue
                break  # Condition met stop asking for another input

            # Modify computer's board accordingly to players chosen coordinate, end turn
            fire_torpedo(computer_board, player_target)

            # Computer's turn
            # Define the range of valid actions (coordinate selections) for computer
            valid_targets = define_valid_targets(player_board)
            # Choose random coordinate from the updated valid range
            computer_target = helper.choose_torpedo_target(player_board, valid_targets)
            # Modify player's board accordingly to players chosen coordinate, end turn
            fire_torpedo(player_board, computer_target)

            # Print the updated game boards after each side make it's move
            print_boards(player_board, computer_board)

            # Check the results of this round, and if any of the conditions are met, terminate the game. Originally,
            # I intended to print informative messages based on the game outcome, such as whether the player won or
            # lost. However, the autotest expects only the game boards to be printed, so I've commented out those
            # sections to pass the test. Despite this, I believe it is important to inform the user that the game has
            # ended before asking them about the next round.
            if not is_fleet_destroyed(player_board) and is_fleet_destroyed(computer_board):
                # print("You Win! All the enemy ships have been destroyed")
                break
            if not is_fleet_destroyed(computer_board) and is_fleet_destroyed(player_board):
                # print("You Lose! All your ships have been destroyed")
                break
            elif is_fleet_destroyed(player_board) and is_fleet_destroyed(computer_board):
                # print("It's a tie! No one wins")
                break

        # When game terminated (exited the loop) ask the player if he wants to start the loop again (start a new game)
        while True:
            play_again = helper.get_input("Do you want to play again? (Y/N): ".strip().upper())
            if play_again == "Y":
                break
            elif play_again == "N":
                return
            else:
                print("Invalid Input")


if __name__ == "__main__":
    main()
```