BBM 103 Assignment 4 Report

Hacettepe University- Department of Computer Engineering

BARIŞ YILDIZ
Student ID: 2210356107
3.01.2023

Contents

Analysis

The program simulates a “Battleships” game.

There are 2 input files for each player, one that contains data for the board of the player, and one that contains the moves that the player makes. The program needs to create player boards out of the board file and make plays for each player according to their moves. Board size is 10x10.

After initializing the boards, the game starts. In each round, first, both boards get printed without giving away the locations of the ships, then Player1 is asked to perform a move. After the move, the proper cell in Player2’s board gets shot. The round continues with printing the boards again and asking Player2 to perform a move. After Player2 shoots a cell in Player1’s board, the round ends. After the round ends, the program checks if any player has lost all their ships. If so, the player who lost all their ships loses the game. If both players have lost all their ships, the game is a draw.

Lastly, the final state of both boards is printed, showing the positions of surviving ships if such ships are present.

If a player hits a ship in their opponent’s board, the “hit” is represented with an “X” symbol. If it is a miss, the hit is represented with an “O” symbol. Also, if a ship’s last surviving part is hit by a move, an “X” is put in the section below the opponent’s board which shows how many of the opponent’s ships are intact.

Any inconvenient cases in the “board” and/or “moves” files should be handled. An exception in the board file should close out the program while an exception in the moves file should only prompt an error message and continue with the next move.

Design

Before Starting

Before starting, we need to check if the input files are in a location that we can access. To do this, we will use try blocks to try to read these files and except blocks to catch IOErrors. In except blocks, we will add the problematic file's name to a list called "**bad_files**" We will do 4 different try-except blocks for each input file to exactly get every problematic file's name. Optional files will not cause any problems, so we will skip those. Lastly, if there are any, we will give an exception stating that we cannot reach the problematic files.

1) Go through each board and moves file and try to read them.
2) Add any file that cannot be reached to the ***bad_files*** list.
3) If ***bad_files*** is not empty, print "IOError: input file/files [contents of ***bad_files***] is/are not reachable. Else, do not print anything and continue the program.

Also, we will define a dictionary "**player_info**" that includes every needed data structure for each player (grid lists for storing hidden boards, ship dictionaries for storing coordinates of ships, ship state lists for the states of ships.)

1) Assign variables ***player1_grid, player2_grid*** to 10 by 10 lists consisting of only "-" character.
2) Assign variables ***player1_boats, player2_boats*** to empty dictionaries.
3) Assign variable ***boat_states*** to a list consisting of 18 "-" characters.
4) Assign variable ***boats*** to a dictionary that maps ship/boat names (first letters of them) to lists that contain a) ship size b) ship count respectively.
5) Assign variable ***player_info*** to a dictionary that maps players to dictionaries that map descriptive strings to data structures associated with the player. For example, ***player_info***["Player1"]["Grid"] will return ***player1_grid***.

Getting the inputs

Boards/Grids: First things first, we need to get the board information and create both players' boards. We can define a function "**open_grid_file**" to read the board files of each player and store all coordinates of the board into a dictionary "**hidden_grids**" that will be used later when we are going to record shots on the boards. Also, in the same function, we can check if the boards were given as 10x10 grids and if the boards include any characters that cannot be recognized as ships. If so, we can handle these unwanted cases. "**open_grid_file**" will be called to read (or try to read) the input files in section **Before Starting**.

```
def open_grid_file(file, player_name):
    """
    Reads the "board/grid" file of a player and stores the board/grid into the "hidden_grids" dictionary.
    Prototype of hidden_grids : keys = names of players, values = dictionaries that represent the player's board. The keys for this
    dictionary are coordinates (e.g. A1, H3) and values are "" if there is no part of a ship at that coordinate or the letter that represents the
    ship if there is a part of a ship there. Raises an AssertionError if any character other than C,B,S,D,P are encountered.
    Parameters:
    :parameter file: the file that represents the board of a player.
    :parameter player_name: player's name
    :returns: does not return anything, only records the board into a dictionary.
    """
```

1) Read *file* and check if *file* has exactly 10 lines, 10 columns and characters no other than [";","","B","C","S","D","P"]. If these conditions are not satisfied, raise an error(kaboom!), and terminate the program.
2) Otherwise, go through each line and split it by semicolons.
3) Calculate the coordinate of each separated character by looking at its line number and column number.
4) Add all the separated parts with their coordinates to *hidden_grids* with *hidden_grids[player_name][coordinates]* = separated part.

Exceptions: Besides the two exceptions mentioned above, there can also be cases such as boats not being placed correctly, boat parts not being connected to each other, boats overlapping with each other etc. To check these, we can define the function "**find_boat_coordinates**".

For ships that are of type "Battleship" and "Patrol Boat", we can ask the user for 2 other files, one for each player, that show the positions of Battleships and Patrol Boats (Optional files). We can then compare the coordinates given in these optional files to the coordinates of the ships in the given board and decide if there is an error.

For other types of ships, we only need to look for sequential placements of ship parts, only one placement for each ship. We can search through the board both vertically (defining and using the function "**vertical_lookup**") and horizontally (defining and using the function "**horizonal_lookup**") to accomplish this. After that, we can compare how many ships we have found to how many ships there actually should be. Lastly, we can compare the symbol count for the ships that we have found to the symbol count in the board in case of disconnected ships.

If there are no exceptions, we can store the coordinates of each ship individually. This will help us later when we need to check if a ship was fully sunken because of a play. The function "**find_boat_coordinates**" is named like so for this functionality.

We will call this function for each boat type and for each player right after reading the board and moves files of players. The returnee of this function will be stored in player_info[player name]["Boats"][boat symbol].

```
def find_boat_coordinates(player_name, ship):
    """
    Attempts to get a list of coordinates for each part of the ship in the board of the player.
    For example, returns the coordinates of the ship "Carrier" for "Player1". This is done in this
    separate function instead of the read board function because of the different placements of Battleship and Patrol
    Boat. Also because there may be wrong placements of ships in the board file. If this function catches any
    exceptions, the game is stopped. (kaBOOM)
    :param player_name: player's name
    :param ship: ship's name. e.g. "Carrier", "Battleship" ...
    :return: a list that stores each coordinate of each part of the ship.
    """
```

1) If ***ship*** is of type “Battleship” or “Patrol Boat”:
    a. Read the “Optional” file of ***player_name***, split each line of it by (;) and extract the origin point and direction of the ships.
    b. Add ***ship***’s size to the origin point in the direction mentioned to get every coordinate of ship on ***player_name***’s board.
    c. Verify if these coordinates are occupied by ***ship*** and if so, store these coordinates into ***individual_boats***. If not, raise an error and terminate the program.
2) Else, if ***ship*** is of another type:
    a. Call the functions ***vertical_lookup*** and ***horizontal_lookup*** and add them up. (These will return ship counts)
    b. Compare the result of this to the count of ***ship*** and check the board again if there are any disconnected ships. If so, raise an error and terminate the program.
    c. Store the coordinates of the ship that’s been found to ***individual_boats***.
3) Return ***individual_boats***

```
def vertical_lookup(individual_boats, boat_size, symbol):
    """
    A function used for searching for the boats Submarine, Destroyer and Carrier vertically.
    If a boat is found vertically, adds that boat's coordinates to the list individual_boats and lastly,
    returns the number of vertical placed boats it has found.
    :param individual_boats: this is a list that is used by find_boat_coordinates function that stores coordinates
    of every instance of a ship. e.g. [[A1,A2,A3,A4], [A6,B6,C6,D7]] for battleships of a player.
    :param boat_size: the size/length of the boat.
    :param symbol: the symbol which represents a part of a certain boat. e.g. "B" for a part of a Battleship.
    :return: the number of vertical placed boats
    """
```

1) Go through each element of ***hidden_grids[player]***, starting from the first column, vertically, and search for subsequent ***symbol***s of length ***boat_size***.
    a. If ***symbol*** was found, increment ***found length*** by 1. Furthermore, if ***found_length*** is equal to ***boat_size*** after the incrementation, store these coordinates into ***individual_boats*** and increment ***boats_found*** by 1. Reset the value of ***found_length*** to 0.
    b. Else, if ***found_length*** is greater than 0, assign it to zero.
2) At the end, return ***boats_found***.

The algorithm for the **horizontal_lookup** function is very similar to **vertical_lookup**, the only difference is that it searches horizontally rather than vertically.

Moves: Secondly, we need to get the move coordinates of each player and store them in a list. We can use this list later when we ask a player to perform a move in each round. We do not need to check for exceptions about moves right now because these exceptions should not crash our program, they should only make the program read the next move. We can handle these exceptions during combat. We can use a function “**get_hit_coordinates**” to read the moves file of each player and do what is mentioned above. “**get_hit_coordinates**” will also be called to read (or try to read) the input files in section **Before Starting**. If successful, the returnee is assigned to player_info[player][“Moves].

```
def get_hit_coordinates(file):
    """
    Reads the file that contains the move coordinates of a player and returns those coordinates as a list.
    :param file: the file that contains the move coordinates of a player
    :return: a list in the form: [coordinate1, coordinate2, coordinate3 ...] = ["5,E", "4,B", "10,E" ...]
    """
```

1) Open the file and store the contents of it in the local variable ***contents***.
2) Split ***contents*** by semicolons and return this separation list.

Printing The Rounds

Rounds consist of two turns, turns consist of printing the current state of the game and asking players for their moves. The printing of the round is going to be done by a function "**display_move**"

Printing the State of the Game: Information about whose turn it is (current player), which round it is, the size of the board, current boards of both players (with shots from their opponent recorded on them), states of ships (surviving or sunken) should all be printed.

The round number and the current player can be easily identified using a loop. In each iteration, we will print the game state and ask the current player for their move. This is going to be a loop that plays the game.

1) While there are no errors, and the players are still making moves:
2) Player1's turn: Call display_move("Player1","Player2",round_number,move of Player1).
3) Player2's turn: Call display_move("Player2","Player1",round_number,move of Player2).
4) Call ***endgame()***. If game is over, break the loop.
5) Else, increment ***round_number*** by 1.

The size of the board is easy to print as it is constant.

Initially, the boards are seen as empty grids and they are modified each turn according to the move of the current player. The boards are stored in a list variable (player_info[player]["Grid"]) for each player and the modification is made there.

States of ships (***boat_states*** data structure) also get modified according to the state of the boards. We can print the state of boards and the ships using a function "**display_player_grids**". This function is called inside the **display_move** function. As **display_move** function contains many other functions, it will be discussed later.

```
def display_player_grids(state="ongoing"):
    """
    Returns a string showing players' boards side by side. Unless the parameter "state" is given as final,
    the boards shown do not give away the positions of the ships, they only show the moves made/shots fired by the opponent.
    If "state" is given as final, shots and positions of boats are showcased.
    :param state: if it is "final", "final board" is shown with the exposed unsunk ships if any are available.
    Otherwise, board is shown with only the shots marked with "O" (if miss) or "X" (if hit)
    :return: both players' boards side by side as a string.
    """
```

1) If ***state*** is equal to "final", reveal the surviving ships on the boards (modify ***player_info[player]["Grid"]*** to show the surviving ships).
2) Initialize a variable called ***out*** that will store the boards and the state of ships section as a string.
3) Zip the two boards together using zip function (they will be zipped row-wise)
4) Add each of the zipped elements to ***out*** with a newline character at the end of each one (to have the boards side by side).
5) Modify the state of ships section with the ships sunken so far and add it to ***out.***
6) Return ***out***.

Asking Players for Their Moves and Modifying Boards: Firstly, the given move must be valid (the rules are given in the User Catalogue section). If not, the user is asked for another move (in other words, we must read the next element of the list containing all moves of the player) until a valid move is given. We can check if the input is valid by defining and using the function "**check_if_move_is_valid**".

After that, we will shoot the grid the move describes, we can define another function "**shoot_grid**" to do this. Using the hidden_grids dictionary, we can decide whether there was a ship at that location. If not, we will only put an "O" mark at that location and move on to the next turn or round. If so, we will put an "X" mark at that location. Furthermore, we will check if we've hit the whole ship because of this move (the last remaining part of the ship). After each hit, we will delete this coordinate from a list containing coordinates of ships (we can define "**shoot_boat**" to help with this). If, after the deletion, there is no more coordinate left for a ship, this means that we have hit the last part of a ship, causing it to sink. We will put an "X" on the corresponding line, denoting that a ship of that type was sunken.

```
def check_if_move_is_valid(player_name, move):
    """
    This function is used to check if the move entered is valid. If it is not, an error message is returned.
    Otherwise, an empty string is returned. The returnee of this function is used in the function display_move().
    :param player_name: player's name (the player that gives the move)
    :param move: the move coordinates
    :return: an error message or an empty string.
    """
```

1. Check that the ***move*** only has 1 comma. Else, create an error message.
2. Check that the row and column values of ***move*** are not empty. If not, create an error message.
3. Check that row is between 1 and 10 and column is between A and J. If not, create an error message.
4. Check that the move hasn't been made already. Otherwise, create an error message.
5. If an error message has been created, return that error message; otherwise, return an empty string.

```
def shoot_grid(player_name, move):
    """
    This function "shoots" a cell/square in a player's board (puts a "X" if there was a part of a ship there and "O" if not).
    :param player_name: the player's name whose board is bombarded at the time of this function's call (the opponent).
    :param move: the location of the shot.
    :return: does not return anything, only mutates the board.
    """
```

1. Define ***target*** as the content of the coordinate ***move*** describes in ***hidden_grids[player_name]***.
2. If ***target*** is “”:
    a. Place an “O” character on the coordinate ***move*** describes in ***player_info[player_name][“Grid”]***.
3. Else:
    a. Place an “X” character on the coordinate ***move*** describes in player_info[player_name][“Grid”].
4. Call the function ***shoot_boat(player_name, move, target).***

```
def shoot_boat(player_name, coordinates, ship):
    """
    Deletes the coordinate of the part of the boat that was sunken from a list that stores coordinates of all boats of a player.
    If there is no coordinate left for a boat after this operation, that means all parts of a certain boat were sunken.
    Therefore, an "X" is put on the part that shows the states of players' boats.
    :param player_name: the player whose ship part has been sunk
    :param coordinates: coordinates of the sunken ship part
    :param ship: uppercase of first letter of the ship's type (symbol of the ship)
    :return: nothing, only modifies the state of ships.
    """
```

1. Look through ***player_info[player_name][“Boats”]*** and find the list value that the ***ship*** key represents.
2. Delete ***coordinates*** from that list, check if the list is empty afterwards.
3. If it is empty, place an “X” on ***ship***’s line in the states of ships section below ***player_name***’s board.

Lastly, we have the function “**display_move**” that connects all these functions with the game. Recall that this function is called during the game loop.

```
def display_move(player_name, opponent_name, round_number, move):
    """
    This function returns : The player's name who is about to make a move, the coordinates of that move,
    the round number, players' boards just before this current move and the states of players' ships.
    :param player_name: player's name
    :param opponent_name: opponent (the player who player_name is going to attack)'s name
    :param round_number: the round number.
    :param move: coordinates of the move "player" makes.
    :return: a string that contains information about the state of players' boards and
    the move that is about to be made.
    """
```

1. Set the variable of ***out*** to be : round_number, the current player, the grid size, the returnee of ***display_player_grids()*** and a message asking for a move from the current player combined.
2. Investigate ***move*** by calling ***check_if_move_is_valid(move),*** if the input is not valid, repeat this step until it becomes valid. Add the returnees of ***check_if_move_is_valid(move)*** to ***out***.
3. Call ***shoot_grid(opponent_name, move).***
4. Return ***out***.

Ending The Game

After each round, the program checks if the game ended. Preferably after each round instead of after a certain player hits a ship because the game might not end because of the potential lack of moves. Also, each round would be a better idea because there is a chance of the game to be a draw.

We can use a function "**endgame**" to detect if the game is over. This function checks both players' ship states and decides that a player has lost if that player's ship states are full of "X" marks (Or has no "-" marks left). If both players have lost, the game is a draw. If only a certain player has lost, their opponent wins. If no one has lost, it will not conclude anything. If the function concludes that the game is over, the game loop gets broken, and the final boards of both players are printed.

```
def endgame():
    """
    Checks if the game is over.
    :return: a string that if the game is over, contains an ending message (draw or win),
    or an empty string if the game is not over.
    """
```

Look at both players' "state of ships" sections. If a player's section is full of "X" marks, determine that that player has lost.

If only one player has lost, return a message congratulating the opponent of that player. Breal from the loop.

If both players lost, return a message saying that the game is a draw. Break from the loop.

If no player has lost, return an empty string. This means that the game is going to continue

To print the final boards, we will use "**display_player_grids**" function and give it the optional argument "final".

Finally, we will define and use the function "**write_file()**" to write the whole game into the output file "Battleship.out"

```
def write_file(content):
    """
    Writes and prints out the game.
    :param content: the string that needs to be printed out and written to the output file.
    :return: nothing, only writes and prints.
    """
```

1. Take the variable ***content*** that gathers all data about the rounds of the game and its final state.
2. Create the file “Battleship.out” and write ***content*** in it.
3. Lastly, print out the ***content*** to the console.

Programmer's Catalogue

The Main Part of the Program

```
try:
    player1_grid_file, player2_grid_file, player1_moves_file, player2_moves_file = sys.argv[1], sys.argv[2], sys.argv[3], sys.argv[4]
    player1_grid, player2_grid = [["-" for i in range(10)] for j in range(10)], [["-" for k in range(10)] for m in range(10)]
    player1_boats, player2_boats = {}, {}
    player1_boat_index, player2_boat_index = {"C": 0, "B": 2, "D": 6, "S": 8, "P": 10}, {"C": 1, "B": 4, "D": 7, "S": 9, "P": 14}
    boat_states = ["-" for g in range(18)]
    hidden_grids = {"Player1": {}, "Player2": {}}
    boats = {"Carrier": [5, 1], "Battleship": [4, 2], "Destroyer": [3, 1], "Submarine": [3, 1], "Patrol Boat": [2, 4]}

    directions = {"right": (0, 1), "down": (1, 0)}
    player_info = {"Player1": {"Grid": player1_grid, "Boats": player1_boats, "Boat_Indexes": player1_boat_index, "Grid_File": player1_gr
    players = ["Player1", "Player2"]
    bad_files = []                                                  # stores any potentially unreachable file.
```

The program starts from here. Firstly, all needed data structures are defined. The most important data structures here are ***hidden_grids*** which stores all coordinates and their values and ***player_info*** which stores every other related data structure about players.

```
for player in players:
    try:
        open_grid_file(player_info[player]["Grid_File"], player)    # reads the board file of players.
    except IOError:                                                 # IOError handling
        bad_files.append(player_info[player]["Grid_File"])          # if any file is unreachable, appends it to the bad_files list.

    try:
        player_info[player]["Moves"] = get_hit_coordinates(player_info[player]["Moves_File"])   # gets the moves for players
    except IOError:
        bad_files.append(player_info[player]["Moves_File"])         # if any file is unreachable, appends it to the bad_files list.

if bad_files:   # if one or more files provided are unreachable, prints a proper error message.
    if len(bad_files) == 1:
        raise IOError(f"IOError: input file {bad_files[0]} is not reachable.")
    else:
        bad_file_names = ", ".join(bad_files)
        raise IOError(f"IOError: input files {bad_file_names} are not reachable.")

for player in players:
    for boat in boats:                  # gets the ship coordinates for each player, also checks whether the boat placements are valid.
        boat_symbol = boat[0]
        player_info[player]["Boats"][f"{boat_symbol}"] = find_boat_coordinates(player, boat)
```

Here, we are checking if the provided files are reachable or not while also reading them if they are reachable. We are also checking if the board is valid. In the for loop at the bottom, we are checking if the ships are placed correctly while also getting their coordinates if they are.

```
    turn = 0
    output = "Battle of Ships Game\n\n"             # "output" stores the whole game.
    while True:                                     # while the game is going
        try:                                        # calculates results for each round
            for player_index in range(1, 3):
                player = f"Player{player_index}"
                opponent = f"Player{(player_index % 2) + 1}"
                output += display_move(player, opponent, turn + 1, player_info[player]["Moves"][turn]) + "\n"
            end_message = endgame()             # checks if the game is over after each round.
            if end_message != "":               # if the game is over, goes here and breaks the loop
                output += end_message
                break
            turn += 1
        except IndexError:     # this except block is mainly used for ending the game if players run out of moves, but no player has lost yet. (if there is such a case).
            break              # changed from except to except IndexError.
    output += display_player_grids("final")
    write_file(output)
except IOError as error:
    write_file(str(error))
except:                 # in case of any missed errors.
    write_file("kaBOOM: run for your life!")
```

This is the game loop. In each iteration, both players take turns, and the program stores players' moves with the game state (boards, state of ships, round number etc.) to a variable called ***output***. When the game is over, the end message and the final state of the game will be added to the value of ***output***. Finally, ***output*** is written to the "Battleship.out" file as well as to the console. Any error that the program didn't think of or cover will terminate it with the output being "kaBOOM: run for your life!"

Functions

```
def open_grid_file(file, player_name):
    """
    Reads the "board/grid" file of a player and stores the board/grid into the "hidden_grids" dictionary.
    Prototype of hidden_grids : keys = names of players, values = dictionaries that represent the player's board. The keys for this
    dictionary are coordinates (e.g. A1, H3) and values are "" if there is no part of a ship at that coordinate or the letter that represents the
    ship if there is a part of a ship there. Raises an AssertionError if any character other than C,B,S,D,P are encountered.
    """
    with open(file) as f:
        lines = f.readlines()
        line_count = len(lines)
        assert line_count == 10     # our boards have to be 10 x 10, if not, the program should kaBOOM.
        for line_index in range(line_count):
            line_contents = lines[line_index].rstrip("\n").split(";")   # contains every character in every row in the player's board.
            assert len(line_contents) == 10
            for square_index in range(len(line_contents)):
                assert line_contents[square_index] in ["B", "C", "S", "D", "P", ""]
                hidden_grids[player_name][f"{chr(65 + square_index)}{line_index + 1}"] = line_contents[square_index]    # initialize the "hidden_grids" dictionary
```

The ***open_grid_file*** function both reads/stores the board file contents and checks if they are valid. Uses assert statements to check if it is valid and for loops to iterate through every coordinate.

```
def get_hit_coordinates(file):
    """
    Reads the file that contains the move coordinates of a player and returns those coordinates as a list.
    """
    with open(file) as f:
        contents = f.read().rstrip(";")    # removes a trailing semicolon (;) if there is such a semicolon
        coordinates = contents.split(";")
    return coordinates
```

The ***get_hit_coordinates*** function reads and stores the moves of players by only using split and read functions.

```python
def find_boat_coordinates(player_name, ship):
    """
    Attempts to get a list of coordinates for each part of the ship in the board of the player.
    For example, returns the coordinates of the ship "Carrier" for "Player1". This is done in this
    separate function instead of the read board function because of the different placements of Battleship and Patrol
    Boat. Also because there may be wrong placements of ships in the board file. If this function catches any
    exceptions, the game is stopped. (kaBOOM)
    """
    (boat_size, boat_count) = boats[ship]
    symbol = ship[0]    # symbol is the character that is representing the boat. e.g. C for Carrier.
    symbols_in_board = list(hidden_grids[player_name].values()).count(symbol)
    boats_found = 0
    individual_boats = []
    if ship in ["Battleship", "Patrol Boat"]:   # finds the coordinates of Battleships and Patrol Boats.
        with open(player_info[player_name]["Optional_File"]) as f:
            for line in f:
                if line[0] == symbol:
                    (origin, direction) = line.split(";")[0][3:], line.split(";")[1]
                    (row, column) = origin.split(",")
                    boat_coordinates = [f"{chr(ord(column) + (i * directions[direction][1]))}{int(row) + (i * directions[direction][0])}" for i in range(boat_size)]
                    for coordinate in boat_coordinates:
                        assert hidden_grids[player_name][coordinate] == symbol
                    boats_found += 1
                    individual_boats.append(boat_coordinates)
    else:   # finds the coordinates of other ships by calling vertical and horizontal lookup functions.
        boats_found = vertical_lookup(individual_boats, boat_size, symbol) + horizontal_lookup(individual_boats, boat_size, symbol)
    assert boats_found == boat_count and symbols_in_board == boats_found * boat_size    # checks if the given board is valid.
    return individual_boats
```

The ***find_boat_coordinates*** function both checks if the ships in the board file are placed correctly and stores their coordinates. It checks for any possible error using assert statements and uses list comprehensions to get sets of coordinates from optional player files.

```python
def vertical_lookup(individual_boats, boat_size, symbol):
    """
    A function used for searching for the boats Submarine, Destroyer and Carrier vertically.
    If a boat is found vertically, adds that boat's coordinates to the list individual_boats and lastly,
    returns the number of vertical placed boats it has found.
    """
    boats_found = 0
    for column_index in range(65, 75):
        column = chr(column_index)
        found_length = 0
        for row in range(1, 11):
            if hidden_grids[player][column+str(row)] == symbol:
                if found_length == boat_size - 1:
                    boat_coordinates = [f"{column}{row - index + 1}" for index in range(boat_size, 0, -1)]
                    individual_boats.append(boat_coordinates)
                    boats_found += 1
                    found_length = 0
                    continue
                found_length += 1
            elif found_length > 0:
                found_length = 0
    return boats_found
```

```python
def horizontal_lookup(individual_boats, boat_size, symbol):
    """
    A function identical to vertical_lookup(), except it finds horizontal placed ships rather than vertical
    ones. Parameters are also identical to vertical_lookup().
    """
    boats_found = 0
    for row in range(1, 11):
        found_length = 0
        for column_index in range(65, 75):
            column = chr(column_index)
            if hidden_grids[player][column+str(row)] == symbol:
                if found_length == boat_size - 1:
                    boat_coordinates = [f"{chr(column_index - index + 1)}{row}" for index in range(boat_size, 0, -1)]
                    individual_boats.append(boat_coordinates)
                    boats_found += 1
                    found_length = 0
                    continue
                found_length += 1
            elif found_length > 0:
                found_length = 0
    return boats_found
```

The ***vertical*** and ***horizontal lookup*** functions search vertically or horizontally placed ships to get the coordinates of them and to verify their correctness. If they find any coordinates, they append it to the local variable ***individual_boats*** of ***find_boat_coordinates*** which returns it later.

```python
def display_move(player_name, opponent_name, round_number, move):
    """
    This function returns : The player's name who is about to make a move, the coordinates of that move,
    the round number, players' boards just before this current move and the states of players' ships.
    """
    out = f"{player_name}'s Move\n\nRound : {round_number}\t\t\t\t\tGrid Size: 10x10\n\n" + display_player_grids() + f"Enter Your Move: {move}\n"
    potential_error_message = check_if_move_is_valid(player_name, move)
    while potential_error_message != "":    # if the current move is invalid, enters this loop which asks for another move until a valid move is given.
        del player_info[player_name]["Moves"][turn]
        move = player_info[player_name]["Moves"][turn]
        out += f"\n{potential_error_message}Enter Your Move: {move}\n"
        potential_error_message = check_if_move_is_valid(player_name, move)
    player_info[player_name]["Performed_Moves"].append(move)
    shoot_grid(opponent_name, move)
    player_info["Player1"]["Boat_State"], player_info["Player2"]["Boat_State"] = [boat_states[0], *boat_states[2:4], boat_states[6], boat_states[8], *boat_states[10:14]],
    return out
```

The display_move function basically returns each turn (1 round = 2 turns) of the game. It uses a while loop to get a new move if the given move is invalid. It calls the ***check_if_move_is_valid***, ***shoot_grid*** and ***display_player_grids*** functions.

```
def check_if_move_is_valid(player_name, move):
    """
    This function is used to check if the move entered is valid. If it is not, an error message is returned.
    Otherwise, an empty string is returned. The returnee of this function is used in the function display_move().
    """
    try:
        count_of_commas = move.count(",")   # this portion looks at errors related with argument numbers/absence or multiplicity of provided rows and columns.
        assert count_of_commas != 0, f"IndexError: Coordinate '{move}' is missing a comma! Please try again."
        assert count_of_commas == 1, f"ValueError: Coordinate '{move}' has too many commas! There should only be 1 comma that separates the row and the column from each other. Please try again."
        (first_coordinate, second_coordinate) = move.split(",")
        assert move not in player_info[player_name]["Performed_Moves"], f"AssertionError: Invalid Operation."
        assert move != ",", f"IndexError: Coordinate '{move}' is missing a required row and column value! Please try again."
        assert first_coordinate != "", f"IndexError: Coordinate '{move}' is missing a required row value! Please try again."
        assert second_coordinate != "", f"IndexError: Coordinate '{move}' is missing a required column value! Please try again."
        message = ""
        # this portion looks at the correctness of provided rows and columns
        try:    # checking the row
            assert 49 <= ord(first_coordinate) <= 57    # checking if the row is among [1, 2, 3, 4, 5, 6, 7, 8, 9]
        except AssertionError:  # if not, error.
            message = f"ValueError: First operand of coordinate '{move}' is not an integer! Please try again."
        except TypeError:   # the code goes here if provided row is not a character. (if it consists of more than 1 character)
            if first_coordinate != "10":    # except for row = 10, all other possibilites should result in an error message
                try:
                    int_test = int(first_coordinate)    # testing if the input is an integer that has more than 1 digit.
                except ValueError:  # if not, it is a string that results in a value error.
                    message = f"ValueError: First operand of coordinate '{move}' is not an integer! Please try again."
                else:   # if it is, this is an "out of range" error.
                    message = "AssertionError: Invalid Operation."
        try:    # checking the column
            assert 65 <= ord(second_coordinate) <= 90   # checking if the column is an uppercase letter
        except (AssertionError, TypeError):     # if not, value error.
            message = f"ValueError: First and second operand of coordinate '{move}' is invalid! Please try again." if message else f"ValueError: Second operand of coordinate '{move}' is not an uppercase letter! Please try again."
        else:   # if it is, verify that it is not out of bounds.
            try:
                assert ord(second_coordinate) <= 74     # checking if the column is out of bounds
            except AssertionError:  # goes here if so
                message = "AssertionError: Invalid Operation."
        assert not message, f"{message}"    # if the message is not empty (if there is an error), raises an error with that message
    except AssertionError as error_message:
        return f"{str(error_message)}\n\n"
    return ""
```

The function ***check_if_move_is_valid*** uses plenty of assert statements to check for the validity of the move. It uses a lot of assert statements purely because the program needs to prompt different error messages for each type of error. The variable ***int_test*** is only used to test if the int() function works on the first coordinate. A variable ***message*** is used for the error messages instead of a real error because the program does not terminate when a move is faulty. It simply asks for the next move/another move.

```
def shoot_grid(player_name, move):
    """
    This function "shoots" a cell/square in a player's board (puts a "X" if there was a part of a ship there and "O" if not).
    """
    (row, column) = int(move.split(",")[0]), move.split(",")[1]
    target = hidden_grids[player_name][f"{column}{row}"]    # the content ("" or first letters of each ship type) of the targeted square.
    if target == "":    # miss
        player_info[player_name]["Grid"][row-1][ord(column) - 65] = "O"     # place "O"
    else:   # hit
        player_info[player_name]["Grid"][row-1][ord(column) - 65] = "X"     # place "X"
        shoot_boat(player_name, f"{column}{row}", target)


def shoot_boat(player_name, coordinates, ship):
    """
    Deletes the coordinate of the part of the boat that was sunken from a list that stores coordinates of all boats of a player.
    If there is no coordinate left for a boat after this operation, that means all parts of a certain boat were sunken.
    Therefore, an "X" is put on the part that shows the states of players' boats.
    """
    boat_list = player_info[player_name]["Boats"]
    for boat_coordinates in boat_list[ship]:
        if coordinates in boat_coordinates:
            del boat_coordinates[boat_coordinates.index(coordinates)]   # deletes the coordinate
            if len(boat_coordinates) == 0:  # if the ship was completely sunken after this shot
                boat_states[player_info[player_name]["Boat_Indexes"][ship]] = "X"
                player_info[player_name]["Boat_Indexes"][ship] += 1
            break
```

The function shoot_grid uses the hidden_grids dictionary to look at the content of target coordinate. It calls shoot_boat if the move is a hit. hhoot_boat uses for loops to look through the “boats” list of the player and uses the del function the delete the coordinate from that list.

```
def display_player_grids(state="ongoing"):
    """
    Returns a string showing players' boards side by side. Unless the parameter "state" is given as final,
    the boards shown do not give away the positions of the ships, they only show the moves made/shots fired by the opponent.
    If "state" is given as final, shots and positions of boats are showcased.
    """
    if state == "final":    # changes the "hidden" board to show any unsunken ships.
        for player_name in players:
            ship_list = player_info[player_name]["Boats"]
            for ship_type in ship_list:
                for ship in ship_list[ship_type]:
                    for coordinate in ship:
                        symbol = hidden_grids[player_name][coordinate]
                        column, row = ord(coordinate[0])-65, int(coordinate[1:]) - 1
                        player_info[player_name]["Grid"][row][column] = symbol
        out = "Final Information\n\nPlayer1's Board\t\t\t\tPlayer2's Board\n  A B C D E F G H I J\t\t  A B C D E F G H I J\n"
    else:       # keeps the hidden board as is.
        out = f"Player1's Hidden Board\t\tPlayer2's Hidden Board\n  A B C D E F G H I J\t\t  A B C D E F G H I J\n"
    for (common_row, rows) in enumerate(zip(player_info["Player1"]["Grid"], player_info["Player2"]["Grid"])):   # prepares a string that shows rows, columns and squares
        (player1_row, player2_row) = " ".join(rows[0]), " ".join(rows[1])
        common_row += 1
        space = "" if common_row == 10 else " "
        out += f"{common_row}{space}{player1_row}\t\t{common_row}{space}{player2_row}\n"
    out += """
Carrier     {}              Carrier     {}
Battleship  {} {}               Battleship  {} {}
Destroyer   {}              Destroyer   {}
Submarine   {}              Submarine   {}
Patrol Boat {} {} {} {}         Patrol Boat {} {} {} {}\n
""".format(*boat_states)    # this is the part that shows which ships have been sunk. Right side for player2, left for player1.
    return out
```

This function returns the boards of both players and the state of their ships side by side. It uses the functions **enumerate** and **zip** to connect the rows of both boards with a common row index. The state of ships can be seen at the end which is formatted from the ***boat_states*** list variable that simulates the format placeholders as elements.

```
def endgame():
    """
    Checks if the game is over.
    """
    player1_lost, player2_lost = player_info["Player1"]["Boat_State"].count("-") == 0, player_info["Player2"]["Boat_State"].count("-") == 0
    if player1_lost and player2_lost:
        return "It is a Draw!\n\n"
    elif player1_lost or player2_lost:
        return "Player2 Wins!\n\n" if player1_lost else "Player1 Wins!\n\n"
    else:
        return ""


def write_file(content):
    """
    Writes and prints out the game.
    """
    with open("Battleship.out", "w", encoding="utf-8") as f:
        f.write(content)
    print(content)
```

The ***endgame*** function checks if the game is over. Player1_lost and player2_lost are Boolean values that get their truth values from the state of ships part of the table.

The ***write_file*** function writes and prints content. Content is given as the whole game, from each round to the end message and final board.

User Catalogue

User Manual

Provide boards and moves for Player1 and Player2 in files. There are no restrictions on the names of these files.

No.	Class of ship	Size	Count	Label
1	Carrier	5	1	CCCCC
2	Battleship	4	2	BBBB
3	Destroyer	3	1	DDD
4	Submarine	3	1	SSS
5	Patrol Boat	2	4	PP

Table 1: Ship Sizes, Counts and Labels

```
1   ;;;;;;C;;;
2   ;;;;B;;C;;;
3   ;P;;;B;;C;P;P;
4   ;P;;;B;;C;;;
5   ;;;;B;;C;;;
6   ;B;B;B;B;;;;;
7   ;;;;;S;S;S;;
8   ;;;;;;;;;D
9   ;;;;P;P;;;;D
10  ;P;P;;;;;;;D
```

Figure 1: Example "Board" File

In the board file, you should make 10x10 boards and separate each square of the board with semicolons (;). There should be 10 lines and 9 semicolons on each line separating either nothing or an uppercase letter which represents a ship.

In the moves file, the moves should be of the format: [row],[column] with row being a number between 1 and 10, column being an uppercase letter between A and J. Moves should be separated with semicolons (;).

```
1   5,E;10,G;8,I;4,C;8,F;4,F;7,A;4,A;9,C;5,G;6,G;2,H;2,F;10,E;3,G;10,I;10,H;4,E;8,G;2,I
    ;4,B;5,F;2,G;10,C;10,B;2,C;3,J;10,A;8,H;4,G;9,E;6,A;7,D;6,H;10,D;6,C;2,J;9,B;3,E;8,
    E;9,I;3,F;7,F;9,D;10,J;3,B;9,F;5,H;3,C;2,D;1,G;7,I;8,D;9,H;7,H;5,J;6,B;4,J;4,I;3,D;
    8,A;2,E;4,H;1,F;10,F;7,B;6,I;1,I;1,E;7,G;7,J;5,C;9,G;6,D;8,J;4,D;1,D;3,I;3,H;1,C;2,
    B;7,C;1,J;
```

Figure 2: Example "Moves" File (Consists of only one line)

Also, provide 2 other files named “OptionalPlayer1.txt” and “OptionalPlayer2.txt”. These files will be used to determine the positions of ships of type “Battleship” and “Patrol Boat” in each player’s boards. The format of each line of these files is: [B1 or B2 or P1 or P2 or P3 or P4]: [uppermost or leftmost index of the ship];[right or down, depending on which way the ship is placed]; Do not change the names of these optional files.

```
1   B1:6,B;right;
2   B2:2,E;down;
3   P1:3,B;down;
4   P2:10,B;right;
5   P3:9,E;right;
6   P4:3,H;right;
```

Figure 3: Example "Optional" File of Player1

```
   A B C D E F G H I J
1  - - - - - - C - - -
2  - - - - B - C - - -
3  - P - - B - C P P -
4  - P - - B - C - - -
5  - - - - B - C - - -
6  - B B B B - - - - -
7  - - - - - S S S - -
8  - - - - - - - - - D
9  - - - - P P - - - D
10 - P P - - - - - - D
```

Figure 4: Board of Player1

Be sure to put these files in the same folder as the program.

Run the program from the terminal with the arguments being board file of player1, board file of player2, moves file of player1, moves file of player2 respectively. You do not need to put the optional files as arguments.

The game will be displayed to the “Battleship.out” file as well as to the terminal.

Restrictions

- The program only accepts 10x10 boards.
- Terminal arguments need to be given in the correct order, otherwise the program will terminate.
- The program terminates if the given board is faulty. (Missing ships, missing semicolons, wrong placement of ships etc.)
- The program terminates if the files are not inside the same folder as the program.
- If the given moves are not enough for the game to end, the program only prints the final state of the boards at the end, without saying that the game was a draw.

Evaluation	Points	Evaluate Yourself / Guess Grading	
Readable Codes and Meaningful Naming	5	5	
Using Explanatory Comments	5	5	
Efficiency (avoiding unnecessary actions)	5	4	
Function Usage	15	15	
Correctness, File I/O	30	25	
Exceptions	20	18	
Report	20	15	
There are several negative evaluations	…	…	