HACETTEPE UNIVERSITY
Department of Computer Science

Course: BBM103 Fall 2022

Assignment 4: Battle of Ships

Mustafa Ege
2210356088

01.01.2023

Analysis

In this assignment, we need to make 2 players play the battle of ships game with some input files. Basically, both players need to put the ships they have to the places they want to put in the beginning of the game, and they need to start attacking their opponent's grid without knowing where the ships are. In order to make the ship sunk, they need to attack every grid the ship lays on. Whoever achieves to make every ship of their opponent sunk, wins.
The input files we need are 2 files for both players for how the ships stays on the grid, 2 files for the attacks of players and 2 more optional files for special ships which are battleships and patrol boats. The reason why they are special is because there is more than one ship in these types, so it is not as easy to know if the ship has sunk or not and we need to approach these ships one by one. Also, while the game is being played, we need to print out every round including what's is going on and what is the current situation. These things should be printed out in both output file and terminal: how the grids of players look without the locations of hidden ships, which move has been made by the player, the status of the round, and the status of the remaining ships. When all the ships of one of the players sunk, the game should end, and the code should print out the winner and the final status of the game including the grids and ship numbers. If all the ships of both players sink in the same round, code should print out the same game status by printing it is a draw. Lastly, we need to check for any exception and error throughout the code.

Design

In this part I explain the general design and the algorithm of the code.

Checking for the I/O Error

There are 4 files which needs to exist for code to properly work ('Player1.txt', 'Player2.txt', 'Player1.in','Player2.in'). Therefore, if one or more of these files don't exist in the same directory of the code, the code raises and i/o error. It checks every file name in a for loop and if it doesn't exist, it adds it to the list. If the list is not empty it raises i/o error and prints out the files' names which create the problem.

Another I/O problem might happen while running the code on the terminal due to non-existence of the parameter that has been given in the python run. Example:
python3 Assignment4.py '.txt' 'Pla2.txt' 'Player1.in' 'Play2.in'
Here in this run, after assignment4.py first, second and fourth parameters do not exist in the directory so the code would print out these files are not reachable.

Creating the Players' Boards

player_board function has one parameter which it gets the input it is going to use from this parameter. Firstly, it seperates the input by the semicolons. Then it adds every item in the list of this separated input into the list. But before doing this it adds hyphen – for every empty square in the player board. Because when we print out the final information, we will need this list.

Then we call this function for both players while using the input files player1.in and player2.in and assign them to 2 different variables.

Additionally, we need 2 more lists just like previous lists, except these lists will be completely made of hyphens '-'. Because we will need these lists to print out the hidden grid without the locations of the ships. To make these lists I used nested list comprehension and assigned them to another variable for hidden grid.

After every attack in each turn the items of these lists will be changed.

Creating the Special Ships

There are 5 types of ships in the game and 3 of them are single ships which means there are only one of them for each player. Therefore, these ships don't create any problems. However, there are 2 different ships which are battleships and patrol boats they need to be treated differently since there are more than one of them for each player. For this, we take extra input from optional files and to make the optional file readable by the code we have a function.

The function first reads the file and transforms every line to a list. Then, for every line it splits from colons and semicolons according to the text. Finally, it adds the items of the list to a dictionary containing the ship name the first square and the position of the ship like right or down.

There is another function for completing the remaining squares of the ship according to the first square of the ship and the position. First it checks if the ship is battleship or patrol boat, and it determines how many squares it will add to the ship. Then if the position is right, it adds squares to the 2nd index of the list, if it is down it adds to the first index of the list. In the end it returns battleships and patrol boats of each player separately. We need to call the first function first and with the result of the first, we call the second one. And we assign the results of these function to 2 variables.

The Number of Ships for Each Player

We need to hold the number of ships each player has while the game is being played. Therefore, for 3 ship types there is 1 ship, for one ship type there are 2 ships, and for one ship type there are 4 ships. A function creates a dictionary for each player, and we assign these dictionaries to some variables to use it while printing the number of ships player has each turn and when we determine how many ships are sunk.

Playing the Game / Attacks by Players

In order to play the game, I created a function which we call in every round. The function gets moves by each player, the player boards, the hidden boards and the round number as parameters. It works by using the current move which is two integers which shows a specific location of a square in the grid. After the finding the exact location which player is intended to attack, the function changes the square's information with O or X. If the square information is a letter like B, D which shows that there is a ship in that location, it changes it to X which means that part of the ship has been hit. If the information is empty hyphen '-' which means there is no ship, it changes the info to 'O' which means attack is a miss. While doing this procedure to the player board, the function also changes the information on the hidden board.

Also, if the location that is being targeted has already been hit, then the function raises an assertion error and prints out the error. Then it continues to work with the next input.

Sinking the Ships

There are a few steps we need to consider for sinking the ships. Firstly, we need to check the player boards to see which locations has been hit and whether they are part of the ships or not.

For carrier, destroyer and submarine it is easy to see if the ship has sunk or not since there is only one ship of each type. The function inside the sunken_ship function checks on how many squares does the ship exist, then returns this value. If this value is different than 0, function returns hyphen '-' to use later in the code for printing the remaining ships. If it is 0, function changes the list containing number of ships the player has for the sunken ship and returns 'X' to later print that that ship has sunk.

For patrol boat and battleship, the procedure is different since there are more than 1 ship for each boat. The function checks the special input dictionaries that we created before and checks how many 'X' there is in each ship. If there are 4 'X' for an individual battleship, the function increases the number of sunken ships by 1, because battleship is 4 square long. And the same thing applies to patrol boat for 2 squares. If number of sunken ships is equal to 2 for battleship and 4 the function changes the list containing number of ships a player has. Finally, it returns the number of sunken ships to print out the number later in the code.

In order to change the elements of special input dictionaries to 'X' when they are hit we need another function. This function firstly enters a loop for looking at the elements of special ship lists. Then with these elements it checks the locations of special ships from the players' empty board whether that square has been hit or not. If it's been hit, it changes the location to 'X' in special input dictionary. At last, we call this function for both players for 2 special ship types.

To make things easier, there is one more function to call every function under this title. We will use this function to change everything after attacks in the last part of the code.

Printing the Grid

To print the grid in every round we need a function, and it should have additional conditions for the final round. Inside of this printing whole round function there is another function for printing the board and the remaining ships. This function inside prints 2 boards side by side and the remaining ships of each player. The main function takes the game winning condition as a parameter, and it prints different things according to this information. If the game hasn't finished, then it prints the round number, players' hidden boards, and the next move at last. If the game has finished and there is a winner, it prints the final information with the open boards which shows every ship location. Finally, if the game has finished but there is no winner it prints that the game's a draw and the open boards.

Game Over Condition

The function checks the dictionary containing the number of ships a player has if the value of some key is greater than one, which means there is at least one ship remaining, it returns none. If the value is zero, which means there is no ship remaining for that player, it returns True.

Running Everything for the Code

Code enters a for loop which lasts for the number of moves by players. But to determine if one of the players run out of moves, it takes the maximum of 2 players' moves.

Firstly, it prints the grid without any attack, then the first player makes the move and attacks, after this it prints the grid again and finally the second player makes the move. If these blocks give error because it is out of moves, it prints which player's moves are over.

To end the game, we call the function which calls every sunken ship function to change the lists and dictionaries which hold the information about remaining ships and sunken ships. After this according to the winning condition, this code block calls the printing function with the suitable parameters which are the parameters of the winning player or the draw game information.

In case anything else happens in the code, it prints kaboom run for life and ends the program.

Programmer's Catalogue

In this section I explain the whole code part by part, there is the whole code I've written, the time I spent for analysis, implementation, testing and how this code can be reused by the others.

```
output = open('Battleship.out','w')
def printer(text):
    output.write(text)
    print(text,end='')
```

In order to print out the information to terminal and to the output file at the same time I created this function.

```
except_list = []
input_list = ['Player1.txt','Player2.txt','Player1.in','Player2.in']
for input_file in input_list:
    if not os.path.exists(input_file):
        except_list.append(input_file)
try:
    if except_list != []:
        raise IOError
except:
    printer('IOError: input file(s) ')
    for file in except_list:
        printer(file+' ')
    printer('is/are not reachable.\n')
    sys.exit()
```

For loop checks if the current directory contains the required files (input list) exist or not and if it doesn't it adds the non-existent ones to the list. As long as the list is not empty it gives io error and it prints the files which do not exist.

```
try:
    player1_start = open(sys.argv[1],'r')
    player2_start = open(sys.argv[2],'r')
    player1_input = open(sys.argv[3],'r')
    player2_input = open(sys.argv[4],'r')
except Exception as e:
    printer(f'IOError: input file(s) {e.filename} is/are not reachable.\n' )
    sys.exit()
```

This code block checks if the terminal run is correct. If the file name has been written wrongly it gives the error that file does not exist, not reachable.

```
def player_board(player_number):
    input_list = player_number.read().splitlines()
    player_map = list()
    for line in input_list:
        with_hyphen = ['-' if i =='' else i for i in line.split(';')]
        player_map.append(with_hyphen)
    return player_map
player1_board = (player_board(player1_start))
player2_board = (player_board(player2_start))
```

This function will take the player.txt input files as the parameter and it will split the squares by semicolons. In for loop, it makes a multi-dimensional list. Lastly it returns the whole board as a list.

```
lettervalue = dict(zip("ABCDEFGHIJ",range(0,10)))
lettervaluereverse = dict(zip(range(0,10), "ABCDEFGHIJ"))
```

Instead of using the letters of squares, I use numbers for the second index of the square. This code gives the number form of the letter from the alphabet.

```
def player_moves(player_number):
    input_list = player_number.read().rstrip(';').split(';')
    moves_list = list()
    for i in input_list:
        try:
            two_digit = i.split(',')
            if len(two_digit) < 2:
                raise IndexError
            elif two_digit[0] == '' or two_digit[1] == '':
                raise IndexError
            second_value = None
            try:
                second_value = int(two_digit[1])
            except:
                pass
            if len(two_digit[0]) <= 2 and len(two_digit[1]) ==1:
                if isinstance(second_value,int):
                    raise ValueError
                elif len(two_digit) >2:
                    raise ValueError
            else:
                raise ValueError

            two_digit[0] = int(two_digit[0])-1
            assert two_digit[0] in range(0,10)
            assert two_digit[1] in ['A','B','C','D','E','F','G','H','I','J']
            two_digit[1] = lettervalue[two_digit[1]]
            moves_list.append(two_digit)
        except ValueError :
            printer(f'ValueError: ({i}) is a wrong input. There should be two values and first value should be a number(integer) and second should be a letter(string) like (4,C)\n\n')
            continue
        except IndexError:
            printer(f'IndexError: ({i}) is a wrong input. It is missing one or more arguments. There should be two values like (4,C)\n\n')
            continue
        except AssertionError:
            printer('AssertionError: Invalid Operation.\n\n')
            continue
    return moves_

player1_moves = player_moves(player1_input)
player2_moves = player_moves(player2_input)
```

This function takes the player.in input files and make them usable for the code by splitting the colons and semicolons. However there might be some wrong inputs so if the input is not in the form of 1,A or 5,B it raises some errors like Index Error and Value Error. It checks the two_digit variable to achieve this. In the index error part it checks if one of the indexes are missing like ,C or ,; . And in value error part it checks if the format is correct or not like 5,5 or A,A or 1,1,A. The last error is Assertion error in this part it checks if the given input inside the grid borders. It shouldn't be more than 10 as number and J as letter. At last it returns the moves list by the player.

! Because I check for the error here in this function, the code prints the error in the beginning of the game instead of the round which causes the problem. The code keeps playing the game after skipping the wrong input but doesn't print in that round.

```
player1_empty_board = [['-' for i in range(10)]for i in range(10)]
player2_empty_board = [['-' for i in range(10)]for i in range(10)]
```

These variables are for printing the empty board without the locations of the ships. They are lists with hyphens '-'.

```
optional_input1 = open('OptionalPlayer1.txt','r')
optional_input2 = open('OptionalPlayer2.txt','r')

def optinal_input(input):
    battleship_dic_list = list()
    for line in input.read().splitlines():
        battheship_dic = dict()
        ship_feature = line.split(':')[1].rstrip(';').split(';')
        index1,index2 =int(ship_feature[0].split(',')[0])-1,lettervalue[ship_feature[0].split(',')[1]]
        ship_feature[0]=[]
        ship_feature[0].append(index1)
        ship_feature[0].append(index2)
        battheship_dic[line.split(':')[0]] = ship_feature
        battleship_dic_list.append(battheship_dic)
    return battleship_dic_list
```

This function is for taking the optional inputs from the text files and making it usable. I needed the optional files to work with the ship types with multiple ships. The function first reads the file then seperates by semicolons and colons and make a dictionary whose keys are the names of ships like B1, P2 and the value is a list with the first square and the position of the ship. [{'B1': [[5, 1], 'right']}, {'B2': [[1, 4], 'down']}, {'P1': [[2, 1], 'down']}…

```
def ship_completer(ships):
    def coordinate_adder(ship_type):
        if ship_type == 'B':
            end = 4
        elif ship_type == 'P':
            end = 2
        if ship[key][1] == 'right':
            for column in range(1,end):
                coordinate = [ship[key][0][0],ship[key][0][1]+column]
                ship[key].append(coordinate)
            ship[key].remove('right')
        elif ship[key][1] == 'down':
            for column in range(1,end):
                coordinate = [ship[key][0][0]+column,ship[key][0][1]]
                ship[key].append(coordinate)
            ship[key].remove('down')

    for ship in ships:
        for key in ship:
            if key[0] == 'B':
                coordinate_adder('B')
            elif key[0] == 'P':
                coordinate_adder('P')
    return ships[0:2],ships[2:6]

player1_battleships,player1_patrolboats = ship_completer(optinal_input(optional_input1))
player2_battleships,player2_patrolboats = ship_completer(optinal_input(optional_input2))
battle_ships = [player1_battleships] + [player2_battleships]
patrol_boats = [player1_patrolboats] + [player2_patrolboats]
```

After making the optional ship dictionaries, this function adds the remaining squares of the ship to that dictionary. If the ship type is battleship which is B for short it enters a loop for 3 times since the ship is 4 squares long and the first square already exists in the dictionary. To understand how to add the squares, function checks the position of the ship. If it is right, it adds the square by increasing the second index which is the letter(column). If if is down same thing with the first index which is the number (row).

To make things easier with more variables, it returns battleships and patrol boats separately. After this we assign the results for each player.

```
player1_ships, player2_ships = dict(),dict()
def ship_list(player_ships):
    for i in ['C','D','S']:
        player_ships[i] = 1
    player_ships['P'] = 4
    player_ships['B'] = 2
ship_list(player1_ships)
ship_list(player2_ships)
players_ships = [player1_ships,player2_ships]
```

This function creates a dictionary holding how many ships each player has to understand when a player has run out of ships later in the code.

```
def attack(movelist,playerboard,playeremptyboard,index):
    global player1_board,player2_board,player1_empty_board,player2_empty_board
    line = movelist[index][0]
    column = movelist[index][1]
    try:
        if playerboard[line][column] != '-':
            assert playerboard[line][column] != 'X' and playerboard[line][column] != 'O'
            playerboard[line][column] = 'X'
            playeremptyboard[line][column] = 'X'
        else:
            assert playerboard[line][column] != 'X' and playerboard[line][column] != 'O'
            playerboard[line][column] = 'O'
            playeremptyboard[line][column] = 'O'
    except AssertionError:
        printer('AssertionError: Invalid Operation.\n\n')
```

The attack function takes the move list, player's hidden and normal board and the round number. It finds the square of the intended attack, then if it is not empty '-' it changes and there is a ship, it changes the info from '-' to 'X' and if it is empty it changes to 'O'. Also if the square has already been hit, then it gives assertion error.

```
def sunken_ship(ship_type,player_number,player_boards=None):
    def remaining_ship_counter(ship_type
        remaining = 0
        for line in player_boards:
            remaining += (line.count(ship_type))
        return remaining
    if ship_type == 'C' or ship_type== 'D' or ship_type =='S':
        if remaining_ship_counter(ship_type) != 0:
            return '-'
        else:
            players_ships[player_number-1][ship_type] = 0
            return 'X'
    elif ship_type == 'B':
        sunk = 0
        for boat in battle_ships[player_number-1]:
            for boatname in boat:
                if boat[boatname].count('X') == 4:
                    sunk += 1
                    if sunk == 2:
                        players_ships[player_number-1][ship_type] = 0
        return sunk
    elif ship_type == 'P':
        sunk = 0
        for boat in patrol_boats[player_number-1]:
            for boatname in boat:
                if boat[boatname].count('X') == 2:
                    sunk += 1
                    if sunk == 4
                        players_ships[player_number-1][ship_type] = 0
        return sunk
```

Sunken_ship function works differently for different ships. If the ship is single ship type it counts the number of ship squares remaining in the player’s board and if it is more than 0 which means the ship still hasn’t sunk, it returns ‘-‘ if it has sunk it returns ‘X’. These are for printing the later in the code. Also if the ship has sunk it changes the dictionary which holds the number of ships player has.
If it is the multiple ships type, then there is the variable called sunk and the function increases this variable by counting the number of that ship type which has sunk. If this variable reaches 4 for battleship it means all the ships of the battleship type has sunk and it changes the dictionary with ship numbers. The way this function checks for the sunken multiple ship types is in the next function.

```
def special_sunken_ship_caller():
    def special_sunken_ship(special_ship_list,player_board):
        for ship in special_ship_list:
            for i in ship:
                for coordinate in ship[i]:
                    if coordinate == 'X':
                        continue
                    if player_board[coordinate[0]][coordinate[1]] == 'X':
                                special_ship_list[special_ship_list.
index(ship)][i][ship[i].index(coordinate)] ='X'
    special_sunken_ship(player1_battleships,player1_empty_board)
    special_sunken_ship(player1_patrolboats,player1_empty_board)
    special_sunken_ship(player2_battleships,player2_empty_board)
    special_sunken_ship(player2_patrolboats,player2_empty_board)
```

Special_sunken_ship function looks for the locations in the battleship and patrolboats list, then with these location it checks the same locations in the player boards. If the location in player board is ‘X’ which means that square has been hit, it changes the battleship and patrol boats list as well. And the previous function sunken_ship checks these lists to understand if the whole ship has been hit or not.

```
def all_sunk_ship_caller(): # calling every remaining ship function in one place
    special_sunken_ship_caller()
    sunken_ship('C',1,player1_board),sunken_ship('C',2,player2_board)
    sunken_ship('B',1),sunken_ship('B',2)
    sunken_ship('D',1,player1_board),sunken_ship('D',2,player2_board)
    sunken_ship('S',1,player1_board),sunken_ship('S',2,player2_board)
    sunken_ship('P',1),sunken_ship('P',2)
```

This function calls every function related with being sunk to make things easier in the last part of the code.

```
def print_grid(player1_empty_board,player2_empty_board,player,condition,round=None):
    def boards
        letters = [' ','A','B','C','D','E','F','G','H','I','J']
        printer('{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<1}\t\t'.format(*letters))
        printer('{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<1}\n'.format(*letters))
        for number in range(10):
            printer('{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<1}\t\t'.format(number+1,*player1_empty_board[number]))
            printer('{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<1}\n'.format(number+1,*player2_empty_board[number]))
        printer('\n')
        battleship_sign1,battleship_sign2 = (sunken_ship('B',1)*'X '+ (2-(sunken_ship('B',1)))*'- '),(sunken_ship('B',2)*'X '+ (2-
(sunken_ship('B',2)))*'- ')
        patrol_sign1, patrol_sign2 = (sunken_ship('P',1)*'X '+ (4-(sunken_ship('P',1)))*'- '),(sunken_ship('P',2)*'X '+ (4-
(sunken_ship('P',2)))*'- ')
        printer('Carrier\t\t{}\t\t\t\tCarrier\t\t{}\n'.format(sunken_ship('C',1,player1_board),sunken_ship('C',2,player2_board)))
        printer('Battleship\t{}\t\t\t\tBattleship\t{}\n'.format(battleship_sign1.rstrip(' '),battleship_sign2.rstrip(' ')))
        printer('Destroyer\t{}\t\t\t\tDestroyer\t{}\n'.format(sunken_ship('D',1,player1_board),sunken_ship('D',2,player2_board)))
        printer('Submarine\t{}\t\t\t\tSubmarine\t{}\n'.format(sunken_ship('S',1,player1_board),sunken_ship('S',2,player2_board)))
        printer('Patrol Boat\t{}\t\t\tPatrol Boat\t{}\n\n'.format(patrol_sign1.rstrip(' '),patrol_sign2.rstrip(' ')))

    if condition == False:
        round_count = round
        printer("{}'s Move\n\n".format(player))
        grid_size = 'Grid Size: 10x10\n\n'
        printer("Round : {}\t\t\t\t\t{}".format(round_count,grid_size))
        p1 = "Player1's Hidden Board"
        p2 = "Player2's Hidden Board\n"
        printer('{}\t\t{}'.format(p1,p2))
        boards()
        printer('Enter your move: {},{}\n'.format(*current_move))
        printer('\n')
    elif condition == True:
        printer(f'{player} Wins!\n\nFinal Information\n\n')
        p1 = "Player1's Board"
        p2 = "Player2's Board\n"
        printer("{}\t\t\t\t{}".format(p1,p2))
        boards()
    elif condition == 'Draw':
        printer(f"It's a draw!\n\nFinal Information\n\n")
        p1 = 'Player1’s Board'
        p2 = 'Player2’s Board\n'
        printer("{}\t\t\t\t{}".format(p1,p2))
        boards()
```

The boards function inside of the function makes the visual representation of the player board with numbers letters and the grid locations. I used string formatting to print out the squares with '-' 'X' and 'O's.
Also the same function prints out the name of ships with the symbols which represents how many ship remains.
Then the print grid function prints different things based on what is going on in the game. If the game hasn't finished yet, it prints hidden boards, rounds and moves. If it has finished and the was won by one player it prints the final info with normal board and tells which player has won. If it is a draw, it prints it is a draw and prints the final boards.

```
def game_over_condition(playership):
    total = 0
    for key in playership:
        total += playership[key]
    if total == 0:
        return True
```

This function checks the dictionary with the number of ships players have. If every key has 0 value it means every ship has sunk so that player loses.

```
try:
    round = 1
    printer('Battle of Ships Game\n\n')
    for move in range(max(len(player1_moves),len(player2_moves))):
        try:
            current_move = [player1_moves[move][0]+1,lettervaluereverse[player1_moves[move][1]]]
            print_grid(player1_empty_board,player2_empty_board,'Player1',False,round)
            attack(player1_moves,player2_board,player2_empty_board,move)
        except IndexError:
            printer("Player 1's moves are over.\n")
            break
        all_sunk_ship_caller
        try:
            current_move = [player2_moves[move][0]+1,lettervaluereverse[player2_moves[move][1]]]
            print_grid(player1_empty_board,player2_empty_board,'Player2',False,round)
            attack(player2_moves,player1_board,player1_empty_board,move)
        except IndexError:
            printer("Player 2's moves are over.\n")
            break
        all_sunk_ship_caller()
        if game_over_condition(player2_ships) and not game_over_condition(player1_ships):
            print_grid(player1_board,player2_board,'Player1',True)
            break
        elif game_over_condition(player1_ships) and not game_over_condition(player2_ships):
            print_grid(player1_board,player2_board,'Player2',True)
            break
        elif game_over_condition(player1_ships) and game_over_condition(player2_ships):
            print_grid(player1_empty_board,player2_empty_board,'','Draw')
            break
        round += 1
except:
    printer('kaBOOM: run for your life!')
```

Here we call every function needed one by one. Firstly it prints the name of the game the enters a loop with the maximum length of player's moves. Inside the loop it firstly prints the grid and the information without any attack then first player makes their move. After this we call the all_sunk_ship_caller to update the ship lists. Then it prints the grid again with the same round number and the second player makes their move. Then we call the all_sunk_ship_caller function again to update, because there has been another attack by the other player.
If any player runs out of moves and there are still moves by the other player, the code handles it by printing the player's moves are over.
If the game finishes before the moves we call the functions for printing the final information and we break the loop to end the game.

```
import sys
import os

output = open('Battleship.out','w')

def printer(text): #function for printing to terminal and to the output file at the same time
    output.write(text)
    print(text,end='')

except_list = []
input_list = ['Player1.txt','Player2.txt','Player1.in','Player2.in']
for input_file in input_list: # if required input files doesn't exist it adds them to the except list
    if not os.path.exists(input_file):
        except_list.append(input_file)

# checks if all of the required files ('Player1.txt','Player2.txt','Player1.in','Player1.in') exists or not
try:
    if except_list != []:
        raise IOError
except:
    printer('IOError: input file(s) {} is/are not reachable.\n'.format(except_list))
    sys.exit()

try:   #checks if terminal input file names exists or not in the file directory
    player1_start = open(sys.argv[1],'r')
    player2_start = open(sys.argv[2],'r')
    player1_input = open(sys.argv[3],'r')
    player2_input = open(sys.argv[4],'r')
except Exception as e:
    printer(f'IOError: input file(s) {e.filename} is/are not reachable.\n' )
    sys.exit()
```

```
def player_board(player_number):
    input_list = player_number.read().splitlines()
    player_map = list() #multi-dimensional list for the first appearance of the player's board
    for line in input_list:
        with_hyphen = ['-' if i =='' else i for i in line.split(';')]
        player_map.append(with_hyphen) #splitting the input with ; semicolons and making
another list
    return player_map

player1_board = (player_board(player1_start))
player2_board = (player_board(player2_start))

lettervalue = dict(zip("ABCDEFGHIJ",range(0,10)))   #assigning numbers to letters from A to J
lettervaluereverse = dict(zip(range(0,10), "ABCDEFGHIJ"))   #assigning numbers to letters from
A to J

def player_moves(player_number):    #function for making a list containing all the moves by a
player
    input_list = player_number.read().rstrip(';').split(';')
    moves_list = list()
    for i in input_list:
        try:    #catches if the input is missing arguments (index error)
            two_digit = i.split(',')
            if len(two_digit) < 2:
                raise IndexError
            elif two_digit[0] == '' or two_digit[1] == '':
                raise IndexError

            second_value = None
            try:
                second_value = int(two_digit[1])
            except:
                pass
            #catches if the input contains values which can not be interpreted by the code
            if len(two_digit[0]) <= 2 and len(two_digit[1]) ==1:
                if isinstance(second_value,int):
                    raise ValueError
                elif len(two_digit) >2:
                    raise ValueError
            else:
                raise ValueError

            two_digit[0] = int(two_digit[0])-1  #converting from str to int
            assert two_digit[0] in range(0,10)
            assert two_digit[1] in ['A','B','C','D','E','F','G','H','I','J']
            two_digit[1] = lettervalue[two_digit[1]] #using integers instead of letters like A B .. J
            moves_list.append(two_digit)
        except ValueError :
            printer(f'ValueError: ({i}) is a wrong input. There should be two values and first value
should be a number(integer) and second should be a letter(string) like (4,C)\n\n')
            continue
        except IndexError:
            printer(f'IndexError: ({i}) is a wrong input. It is missing one or more arguments. There
should be two values like (4,C)\n\n')
            continue
        except AssertionError:
            printer('AssertionError: Invalid Operation.\n\n')
            continue
    return moves_list #moves_list is a nested list to have the player's moves

player1_moves = player_moves(player1_input)
player2_moves = player_moves(player2_input)

player1_empty_board = [['-' for i in range(10)]for i in range(10)] #empty lists without the
locations of ships
player2_empty_board = [['-' for i in range(10)]for i in range(10)]

optional_input1 = open('OptionalPlayer1.txt','r')
optional_input2 = open('OptionalPlayer2.txt','r')

# function to make list for battleships and patrol boats but this function only records the first
square and the position of the boat
def optinal_input(input):
    battleship_dic_list = list()
    for line in input.read().splitlines():
        battheship_dic = dict()
        ship_feature = line.split(':')[1].rstrip(';').split(';')
        index1,index2 =int(ship_feature[0].split(',')[0])-1,lettervalue[ship_feature[0].split(',')[1]]
        ship_feature[0]=[]
        ship_feature[0].append(index1)
        ship_feature[0].append(index2)
        battheship_dic[line.split(':')[0]] = ship_feature
        battleship_dic_list.append(battheship_dic)
    return battleship_dic_list

def ship_completer(ships): #adds the remaning coordinates of ships like patrolboat and battleship
to the list
    def coordinate_adder(ship_type):
        if ship_type == 'B':
            end = 4
        elif ship_type == 'P':
            end = 2
        if ship[key][1] == 'right':
            for column in range(1,end):
                coordinate = [ship[key][0][0],ship[key][0][1]+column]
                ship[key].append(coordinate)
            ship[key].remove('right')
        elif ship[key][1] == 'down':
            for column in range(1,end):
                coordinate = [ship[key][0][0]+column,ship[key][0][1]]
                ship[key].append(coordinate)
            ship[key].remove('down')

    for ship in ships:
        for key in ship:
            if key[0] == 'B':
                coordinate_adder('B')
            elif key[0] == 'P':
                coordinate_adder('P')
    return ships[0:2],ships[2:6]

player1_battleships,player1_patrolboats = ship_completer(optinal_input(optional_input1))
player2_battleships,player2_patrolboats = ship_completer(optinal_input(optional_input2))
battle_ships = [player1_battleships] + [player2_battleships]
patrol_boats = [player1_patrolboats] + [player2_patrolboats]

player1_ships, player2_ships = dict(),dict()
def ship_list(player_ships): #creating a dictionary to hold how many ships a player has
    for i in ['C','D','S']:
        player_ships[i] = 1
    player_ships['P'] = 4
    player_ships['B'] = 2
ship_list(player1_ships)
ship_list(player2_ships)
players_ships = [player1_ships,player2_ships]

def attack(movelist,playerboard,playeremptyboard,round): #to attack a square on the grid and
change the info of that square
    global player1_board,player2_board,player1_empty_board,player2_empty_board
    line = movelist[round][0]
    column = movelist[round][1]
    try:
        if playerboard[line][column] != '-':
            assert playerboard[line][column] != 'X' and playerboard[line][column] != 'O'
            playerboard[line][column] = 'X'
            playeremptyboard[line][column] = 'X'
        else:
            assert playerboard[line][column] != 'X' and playerboard[line][column] != 'O'
            playerboard[line][column] = 'O'
            playeremptyboard[line][column] = 'O'
    except AssertionError:
        printer('AssertionError: Invalid Operation.\n\n')

#to determine if the ship type has sunk or not and it works differently if the ship type has more
than one ship (battleship, patrol boat)
def sunken_ship(ship_type,player_number,player_boards=None):
    def remaining_ship_counter(ship_type): # counts the number of the ship type in whole player
board
        remaining = 0
        for line in player_boards:
            remaining += (line.count(ship_type))
        return remaining
    if ship_type == 'C' or ship_type== 'D' or ship_type =='S':
        if remaining_ship_counter(ship_type) != 0:
            return '-'
        else:
            players_ships[player_number-1][ship_type] = 0
            return 'X'
    elif ship_type == 'B':
        sunk = 0
        for boat in battle_ships[player_number-1]:
            for boatname in boat:
                if boat[boatname].count('X') == 4: # checks if 4 of the squares has sunk or not of the
battleship
                    sunk += 1
                    if sunk == 2:  # if 2 boat has sunk it makes the number of the player's ship to zero
                        players_ships[player_number-1][ship_type] = 0
        return sunk
    elif ship_type == 'P':
        sunk = 0
        for boat in patrol_boats[player_number-1]:
            for boatname in boat:
                if boat[boatname].count('X') == 2: # checks if 2 of the squares has sunk or not of the
patrol boat
                    sunk += 1
                    if sunk == 4: # if 4 boat has sunk it makes the number of the player's ship to zero
                        players_ships[player_number-1][ship_type] = 0
        return sunk

def special_sunken_ship_caller():  # to change the special ships' information according to player
boards
    def special_sunken_ship(special_ship_list,player_board):
        for ship in special_ship_list:
            for i in ship:
```

```
            for coordinate in ship[i]:
                if coordinate == 'X':
                    continue
                if player_board[coordinate[0]][coordinate[1]] == 'X':
                    special_ship_list[special_ship_list.index(ship)][i][ship[i].index(coordinate)] ='X'
    special_sunken_ship(player1_battleships,player1_empty_board)
    special_sunken_ship(player1_patrolboats,player1_empty_board)
    special_sunken_ship(player2_battleships,player2_empty_board)
    special_sunken_ship(player2_patrolboats,player2_empty_board)

def all_sunk_ship_caller(): # calling every remaining ship function in one place
    special_sunken_ship_caller()
    sunken_ship('C',1,player1_board),sunken_ship('C',2,player2_board)
    sunken_ship('B',1),sunken_ship('B',2)
    sunken_ship('D',1,player1_board),sunken_ship('D',2,player2_board)
    sunken_ship('S',1,player1_board),sunken_ship('S',2,player2_board)
    sunken_ship('P',1),sunken_ship('P',2)

def print_grid(player1_empty_board,player2_empty_board,player,condition,round=None):
    def boards(): #  prints the grid with letters and numbers in the corners and x and o's later it
prints remaining ships
        letters = [' ','A','B','C','D','E','F','G','H','I','J']
        printer('{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<1}\t\t'.format(*letters))
        printer('{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<1}\n'.format(*letters))
        for number in range(10):

printer('{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<1}\t\t'.format(number+1,*pl
ayer1_empty_board[number]))

printer('{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<2}{:<1}\n'.format(number+1,*pla
yer2_empty_board[number]))
        printer('\n')
        battleship_sign1,battleship_sign2 = (sunken_ship('B',1)*'X '+ (2-(sunken_ship('B',1)))*'-
'),(sunken_ship('B',2)*'X '+ (2-(sunken_ship('B',2)))*'- ')
        patrol_sign1, patrol_sign2 = (sunken_ship('P',1)*'X '+ (4-(sunken_ship('P',1)))*'-
'),(sunken_ship('P',2)*'X '+ (4-(sunken_ship('P',2)))*'- ')

printer('Carrier\t\t{}\t\t\t\tCarrier\t\t{}\n'.format(sunken_ship('C',1,player1_board),sunken_ship('
C',2,player2_board)))
        printer('Battleship\t{}\t\t\t\tBattleship\t{}\n'.format(battleship_sign1.rstrip('
'),battleship_sign2.rstrip(' ')))

printer('Destroyer\t{}\t\t\t\tDestroyer\t{}\n'.format(sunken_ship('D',1,player1_board),sunken_shi
p('D',2,player2_board)))
```

```
printer('Submarine\t{}\t\t\t\tSubmarine\t{}\n'.format(sunken_ship('S',1,player1_board),sunken_s
hip('S',2,player2_board)))
        printer('Patrol Boat\t{}\t\t\tPatrol Boat\t{}\n\n'.format(patrol_sign1.rstrip('
'),patrol_sign2.rstrip(' ')))

    if condition == False: # as long as there is no winner this block works
        round_count = round
        printer("{}'s Move\n\n".format(player))
        grid_size = 'Grid Size: 10x10\n\n'
        printer("Round : {}\t\t\t\t\t{}".format(round_count,grid_size))
        p1 = "Player1's Hidden Board"
        p2 = "Player2's Hidden Board\n"
        printer('{}\t\t{}'.format(p1,p2))
        boards()
        printer('Enter your move: {},{}\n'.format(*current_move))
        printer('\n')
    elif condition == True: # when there is one winner this block works
        printer(f'{player} Wins!\n\nFinal Information\n\n')
        p1 = "Player1's Board"
        p2 = "Player2's Board\n"
        printer("{}\t\t\t\t{}".format(p1,p2))
        boards()
    elif condition == 'Draw': # when both player wins this block works
        printer(f"It's a draw!\n\nFinal Information\n\n")
        p1 = 'Player1’s Board'
        p2 = 'Player2’s Board\n'
        printer("{}\t\t\t\t{}".format(p1,p2))
        boards()

def game_over_condition(playership): #checks if the players are out of ships to determine if the
game is over
    total = 0
    for key in playership:
        total += playership[key]
    if total == 0:
        return True

try: # the code block for calling every function we need to use
    round = 1
    printer('Battle of Ships Game\n\n')
    for move in range(max(len(player1_moves),len(player2_moves))):
        try:
            current_move =
[player1_moves[move][0]+1,lettervaluereverse[player1_moves[move][1]]]
```

```
            print_grid(player1_empty_board,player2_empty_board,'Player1',False,round) #prints the
first part of the round (player 1 turn)
            attack(player1_moves,player2_board,player2_empty_board,move)
        except IndexError:
            printer("Player 1's moves are over.\n")
            break
        all_sunk_ship_caller() #updates the ships after the first attack
        try:
            current_move =
[player2_moves[move][0]+1,lettervaluereverse[player2_moves[move][1]]]
            print_grid(player1_empty_board,player2_empty_board,'Player2',False,round) #prints the
second part of the round (player 2 turn)
            attack(player2_moves,player1_board,player1_empty_board,move)
        except IndexError:
            printer("Player 2's moves are over.\n")
            break
        all_sunk_ship_caller() #updates the ships after the second attack
        if game_over_condition(player2_ships) and not game_over_condition(player1_ships): #
breaks the loop if there is winning or draw condition
            print_grid(player1_board,player2_board,'Player1',True)
            break
        elif game_over_condition(player1_ships) and not game_over_condition(player2_ships):
            print_grid(player1_board,player2_board,'Player2',True)
            break
        elif game_over_condition(player1_ships) and game_over_condition(player2_ships):
            print_grid(player1_empty_board,player2_empty_board,'','Draw')
            break
        round += 1
except:
    printer('kaBOOM: run for your life!')
```

Time Spent on This Assignment

Time Spent for Analysis	Analysis part for me includes understanding what is wanted and how to do it. I thought of the ways I can get the input from the files and the ways to use them. The best way I could find was using 2 list indexes for the exact location of the ships. And the hardest part was to find how to approach to ship types with multiple ships. I had to use the optional files. This part took around 5 hours

Time Spent for Design and Implementation	This part was the most time-consuming part. Finding an algorithm and coding that algorithm for what I mentioned in the analysis part took a while. I took long time understanding the logic but after I got what I was actually doing, it became easier, and I got faster. This part took around 20-25 hours.
Time Spent for Testing and Reporting	While implementing the code, I tested every step but after I finished everything, I had some small issues that I had to solve, also I needed to add the exception handling. These took around 5 hours for me to solve. For the reporting part I wrote this report in 5-6 hours.

User Catalogue

To use this program as a user only thing you need to do is changing the input files.

This is how player1.txt looks like:

```
;;;;;;C;;;
;;;;B;;C;;;
;P;;;B;;C;P;P;
;P;;;B;;C;;;
;;;;B;;C;;;
;B;B;B;B;;;;;
;;;;;S;S;S;;
;;;;;;;;;D
;;;;P;P;;;;D
;P;P;;;;;;;D
```

As you can see semicolons separate the ship squares and make a grid. So in order to place the ships in different locations you need to change the letters stay on these txt files.

This is how player1.in looks like:

```
10,A;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;
```

As you can see moves are separated with semicolons, while the exact location of the squares is separated by colons. You need to change these .in files according to the place you want to attack your opponent. First index(number) shows the line, and the second index (letter) shows the column.

This is how optionalplayer1.txt looks like:

```
B1:6,B;right;
```

B2:2,E;down;
P1:3,B;down;
P2:10,B;right;
P3:9,E;right;
P4:3,H;right;

This is the last step to consider. For the ship types with multiple ships, you need to define how the ship is located. The first item is for the ships name which goes like the first letter and the number. Then you need to define the first square of the ship and lastly the position of the ship. Even though the program checks for exceptions and errors you need to be cautious with the inputs.

Evaluation	Points	Evaluate Yourself / Guess Grading
Readable Codes and Meaningful Naming	5	5
Using Explanatory Comments	5	5
Efficiency (avoiding unnecessary actions)	5	5
Function Usage	15	15
Correctness, File I/O	30	30
Exceptions	20	18
Report	20	17
There are several negative evaluations	……	