HACETTEPE UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

BBM 103 ASSIGNMENT 4 REPORT

Ahmet İkbal Kayapınar

2210356043

04/01/2023

DESIGN

There will be a function in the program that creates boards using txt files.

The program will store the boards in multi-dimensional lists.

There will be a function to find ships on boards.

The program will store ships in a nested dictionary.

There will be a function in the program that will read the in files.

There will be a function in the program to be able to play one round of the game. This function will mark the hit square, if the ship was hit it will check if it sank.

Every time the program runs without errors, it will pass the round to the other player.

Thanks to the try-except blocks, the program will give an error message and stop in every error that will prevent the program from running. In the event that will affect the rounds, it will skip that part thanks to the try block that only includes that round.

Using the functions in the program, it will create a table showing the current situation at the beginning of each round.

All tables and error messages in the program will be added to the output. Output will be printed to out file and console.

ANALYSIS

In the assignment, we are asked to write a simple battleship game.

The names of the files to be used in the input and output will be retrieved via console arguments.

The boards on which the game will be played will be retrieved via txt files.

Players' moves will be retrieved via in files.

In case of things that may cause an error in the program and in conflict with the rules of the game, the program will add the error message to the output and close.

Only errors that may affect the current round will continue until the round is played without errors and the error message will be added to the output.

The program will detect the ships itself.

Each player will pass the round to the other after playing without errors.

At the end of each round, a table showing the current status of the players' hidden tables will be added to the output.

Sinking ships will be shown in the table.

The program will determine the winner.

The program will print the output both to the out file and to the console.

SOLUTIONS

1. Functions

```
def create_board(file):
    board = list()
    with open(file) as f:
        line_list = f.read().split("\n")
    for line in line_list:
        row = list()
        squares = line.split(";")
        for square in squares:
            if(square=="C" or square=="B" or square=="D" or square=="S" or square=="P"):
                row.append(square)
            elif(square==""):
                row.append("-")
            else:
                raise GenericError
        assert len(row) == 10
        board.append(row)
    assert len(board) == 10
    return board
```

It creates a board using the data it receives from the input files, and returns this board as a multi-dimensional list. It puts the lines of the file into the for loop by making them elements of the list. It splits each row into parts using the *split()* method and saves the letters in a list named *row*. If an incorrect entry is given, it will throw an error. It creates a multi-dimensional list by adding each row list to the *board*.

```
def read_in_files(file):
    with open(file) as f:
```

```
        data = f.read().split(";")[:-1]
    return data
```

It opens the given file. It splits it into parts using the *split()* method and saves it to the *data* variable, excluding the empty part at the end. It returns the *data* variable.

```
def write_output(file, output):
    with open(file, "w") as f:
        f.write(output)
```

It opens the given file and saves the created output to the file.

```
def default_ship_counts(ship_type):
    if(ship_type=="C" or ship_type=="D" or ship_type=="S"):
        return 1
    elif(ship_type=="B"):
        return 2
    elif(ship_type=="P"):
        return 4
```

It keeps the default ship numbers. It returns the required count of the given ship type.

```
def find_ship_length(ship_type):
    if(ship_type=="C"):
        return 5
    elif(ship_type=="B"):
        return 4
    elif(ship_type=="D" or ship_type=="S"):
        return 3
    elif(ship_type=="P"):
        return 2
```

It stores the ship lengths. It returns the length of the given ship type.

```
def find_ships(board):
```

```
    ships = {"C":dict(), "B":dict(), "D":dict(), "S":dict(), "P":dict()}
    ship_positions = list()
    for i in range(len(board)):
        j = 0
        while j<len(board[i]):
            if(j not in ship_positions):
                if(board[i][j]!="-"):
                    ship_length = find_ship_length(board[i][j])
                    ship_count = len(ships[board[i][j]])
                    try:
                        for k in range(j + 1, j + ship_length):
                            if(board[i][k]!=board[i][j]):
                                raise ShipError
                        ships[board[i][j]][ship_count]= [[i], list(range(j, j + ship_length)),
ship_length]
                        j += ship_length - 1
                    except:
                        try:
                            for k in range(i + 1, i + ship_length):
                                if(board[k][j]!=board[i][j]):
                                    raise ShipError
                            ships[board[i][j]][ship_count] = [list(range(i, i + ship_length)), [j],
ship_length]
                            for l in range(ship_length):
                                ship_positions.append(j)
                        except:
                            pass
            j += 1
        for number in range(len(board)):
            try:
                ship_positions.remove(number)
```

```
        except:
            pass
    for ship_type in ships:
        assert len(ships[ship_type]) == default_ship_counts(ship_type)
    return ships
```

It finds the ships, saves the properties of the ships in a nested dictionary named *ships* and returns it. It looks at each square on the board using loops. When it finds a non-empty square, it first checks whether there are as many identical letters as the length of the ship in the horizontal plane using try blocks. If it finds it, it saves the ship's properties as a list in the dictionary whose key is the ship type. It then loops forward so it doesn't check for the same squares again. If it can't find a ship in the horizontal plane, it checks in the vertical plane. If it finds the ship, it saves it as in the previous case. In order to bypass the squares where the ship is located, it records the column number as much as the count that the ship needs to bypass in the list named *ship_positions*. In this way, it checks this list with an if statement in each loop and skips if this column is present in the list. After each pass, it subtracts the number of that column number from the list by one. This will bypass this column once determined.

```
def did_it_sink(ships, ship_type, health, ship):
    if(health==0):
        ships[ship_type].pop(ship)
    if(len(ships[ship_type])==0):
        ships.pop(ship_type)
```

It checks if the ship is sunk. If the ship is sunk, it removes the ship from the dictionary. If there is no ship of that ship type left, it deletes that ship type from the dictionary.

```
def play(board, ships, coordinates):
    r = int(coordinates[0]) - 1
    c = alphabet.index(coordinates[1])
    assert 0<=r<10 and c<10
    square = board[r][c]
    if(square=="-"):
        board[r][c] = "O"
    elif(square=="O" or square=="X"):
        None
```

```
    else:
        board[r][c] = "X"
        try:
            for ship in ships[square]:
                for horizontal in ships[square][ship][0]:
                    if(horizontal==r):
                        for vertical in ships[square][ship][1]:
                            if(vertical==c):
                                ships[square][ship][2] -= 1
                                did_it_sink(ships, square, ships[square][ship][2], ship)
                                raise RuntimeError
        except RuntimeError:
            pass
```

It is the function that enables each round of the game to be played. It takes the coordinates to hit as arguments. If the square to be shot is empty, it assigns the letter *O*, which indicates that the empty place has been hit, to the index representing the square. If there is a ship in the frame to be hit, it replaces the frame with an *X*, scans the ship dictionary using loops and finds the hit ship. Decreases the ship's health and checks if the ship has sunk.

```
def create_row(board, i):
    row = "{:<2,d}".format(i + 1)
    for j in range(len(board[i])):
        if(board[i][j]=="O" or board[i][j]=="X"):
            row += "{} ".format(board[i][j])
        else:
            row += "{} ".format("-")
    row.rstrip()
    return row
```

It creates and returns each row of the players' hidden tables. It checks the hit frames and adds them to the *row* variable. If the square is not hit, it will show as an empty square. It returns the *row* variable.

```
def create_table(board1, board2):
```

```
    table = "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 i in range(len(board1)):
        table += create_row(board1, i) + "\t\t" + create_row(board2, i) + "\n"
    return table + "\n"
```

It creates and returns hidden tables of players'.

```
def ship_counter(ships, ship_type):
    try:
        return len(ships[ship_type])
    except KeyError:
        return 0
```

It finds and returns the number of surviving ships of the given ship type.

```
def ship_chart(ships, ship_type):
    ship_count = ship_counter(ships, ship_type)
    default_ship_count = default_ship_counts(ship_type)
    return ((default_ship_count - ship_count) * "X " + ship_count * "- ").rstrip()
```

It creates the chart showing the number of sunken and surviving ships of the given ship type. It returns the chart.

```
def create_ship_table(ships1, ships2):
    c1 = ship_chart(ships1, "C")
    b1 = ship_chart(ships1, "B")
    d1 = ship_chart(ships1, "D")
    s1 = ship_chart(ships1, "S")
    p1 = ship_chart(ships1, "P")
    c2 = ship_chart(ships2, "C")
    b2 = ship_chart(ships2, "B")
    d2 = ship_chart(ships2, "D")
    s2 = ship_chart(ships2, "S")
```

```
    p2 = ship_chart(ships2, "P")
    table = """Carrier\t\t{}\t\t\t\tCarrier\t\t{}\nBattleship\t{}\t\t\t\tBattleship\t{}
Destroyer\t{}\t\t\t\tDestroyer\t{}\nSubmarine\t{}\t\t\t\tSubmarine\t{}
Patrol Boat\t{}\t\t\tPatrol Boat\t{}""".format(c1, c2, b1, b2, d1, d2, s1, s2, p1, p2)
    return table
```

It creates and returns the table showing the status of the players' ships.

```
def create_output(board1, board2, ships1, ships2):
    table = create_table(board1, board2)
    ship_table = create_ship_table(ships1, ships2)
    return table + ship_table
```

It generates the output of each round and returns the output using the tables created using the previous functions.

2. Main Program

2.1 The While Loop

```
while True:
    if(turn==1):
        ...
    if(turn==2):
        ...
    round_counter += 1
```

Inside the loop, there are if statements that will work on the turns of the players. The program increases the round counter by one after the rounds are completed.

2.2 Turns

```
command = input1[player1_counter]
try:
    coordinates = command.split(",")
    output += "Enter your move: {}\n".format(command)
    play(board2, ships2, coordinates)
```

```
    table = create_output(board1, board2, ships1, ships2)

    intro = "\nPlayer{:d}'s Move\n\nRound : {:d}\t\t\t\t\tGrid Size: 10x10\n\n".format(2,
round_counter)

    output += intro + table + "\n\n"

    turn = 2

    if(len(ships2)==0):

        win += 1
```

The program splits the command into its coordinates with the *split()* method. If it can play the game smoothly with the play function, it creates the table with the *create_output()* function and adds it to the output. It then passes the round to the other player. At the end, it checks whether the player has won the game or not. (As each player's turn program is similar, player1's turn program is used as an example.)

2.3 Win Checker

```
if(win==1):

    winner = "Player1 Wins!"

elif(win==-1):

    winner = "Player2 Wins!"

else:

    winner = "Player1 Wins!, Player2 Wins!, It is a Draw!"
```

It checks if that player won the game in each player's turn. If player1 wins, it adds 1 to the *win* variable, if player2 wins, it subtracts 1 from the *win* variable. Using this information, after the loop ends, this program uses if statements to determine the winner.

3. Exceptions

3.1 General Errors

```
try:

    ...

except IOError:

    correct_inputs = ["Assignment4.py", "Player1.txt", "Player2.txt", "Player1.in",
"Player2.in"]

    incorrect_inputs = [correct_inputs[i] for i in range(1,5) if sys.argv[i]!=correct_inputs[i]]

    output += "IOError: input file(s) {} is/are not reachable.".format(",
".join(incorrect_inputs))
```

```
except:
    output += "kaBOOM: run for your life!"
```

If there is an error that will prevent the game from starting, this error goes to exceptions thanks to the main try block and the program closes. If the input files aren’t correct, the program compares the given names with the list containing the correct file names, finds the wrong arguments and adds them to the output. If there is another general error, the program adds the error message to the output.

3.2 Turn Errors

```
try:
    ...
except IndexError:
    if(len(coordinates)==1):
        if(coordinates[0]==""):
            message = "This command doesn't have any arguments"
        else:
            message = "This command has just one argument"
    else:
        if(coordinates[0]==""):
            if(coordinates[1]==""):
                message = "This command's arguments are empty"
            else:
                message = "This command's first argument is empty"
        else:
            message = "This command's second argument is empty"
    output += "IndexError: {}.\n".format(message)
    round_counter -= 1
except ValueError:
    try:
        int(coordinates[0])
    except:
        try:
```

```
            alphabet.index(coordinates[1])
        except:
            message = "First argument is a non-numeric value and second argument isn't a letter"
        else:
            message = "First argument is a non-numeric value"
    else:
        try:
            alphabet.index(coordinates[1])
        except:
            message = "Second argument isn't a letter"
    output += "ValueError: {}.\n".format(message)
    round_counter -= 1
except AssertionError:
    output += "AssertionError: Invalid Operation.\n"
    round_counter -= 1
finally:
    player_counter += 1
```

If the program returns index error or value error, it checks each *coordinate* for each state using if statements and adds the appropriate error message to the output. If the program returns an assertion error, it adds the error message to the output. Every time the program gives an error, it takes the round counter back by one, ensuring that the game stays in the same round until it is played without any errors. In the finally block, the program moves on to the player's next move at each play attempt.

PROGRAMMER'S CATALOGUE

It took me 2 hours in total to analyze the problem, but in the later stages of the assignment, I saw that the things I analyzed were different and I had to make changes. It took me 3 hours to divide the program into small solutions and develop the main solution. It took me 12 hours to implement each solution and fix the problems. The extra time I spent on making the code more efficient and clean made the assignment take longer than I expected. Although I spent 1 hour after the homework was finished to test it, the tests I did regularly while working on the program also took me 1-2 hours. I wrote the report in 4 hours.

All my code is here:

```
import sys
```

```
# It is for the ship finder. If there is no ship in the checked square, it will give this error.
class ShipError(Exception):
    pass

# It is the error type that used for errors other than specified errors.
class GenericError(Exception):
    pass

def create_board(file):
    board = list()
    # It opens the file, splits it into lines and, saves into a list.
    with open(file) as f:
        line_list = f.read().split("\n")
    for line in line_list:
    # For each line, it creates the row and saves into the list.
        row = list()
        squares = line.split(";")
        for square in squares:
            if(square=="C" or square=="B" or square=="D" or square=="S" or square=="P"):
                row.append(square)
            elif(square==""):
                row.append("-")
            # It throws an error if there is a character other than expected.
            else:
                raise GenericError
        assert len(row) == 10
        board.append(row)
    assert len(board) == 10
    return board
```

```
def read_in_files(file):
    # It opens the file, splits each command and, saves all lines except the last blank line as a list.
    with open(file) as f:
        data = f.read().split(";")[:-1]
    return data


def write_output(file, output):
    with open(file, "w") as f:
        f.write(output)


def default_ship_counts(ship_type):
    if(ship_type=="C" or ship_type=="D" or ship_type=="S"):
        return 1
    elif(ship_type=="B"):
        return 2
    elif(ship_type=="P"):
        return 4


def find_ship_length(ship_type):
    if(ship_type=="C"):
        return 5
    elif(ship_type=="B"):
        return 4
    elif(ship_type=="D" or ship_type=="S"):
        return 3
    elif(ship_type=="P"):
        return 2


def find_ships(board):
```

```
    # Returns a dict that stores dicts whose keys are ship types and which hold the ship's properties.
    ships = {"C":dict(), "B":dict(), "D":dict(), "S":dict(), "P":dict()}
    # It stores column numbers where ships are found.
    # Each number's count represents how many rows later that column will be empty.
    ship_positions = list()
    for i in range(len(board)):
        j = 0
        while j<len(board[i]):
            # It checks if there is currently a ship in that square.
            if(j not in ship_positions):
                if(board[i][j]!="-"):
                    ship_length = find_ship_length(board[i][j])
                    ship_count = len(ships[board[i][j]])
                    # This try block checks if there is a horizontally positioned ship.
                    try:
                        for k in range(j + 1, j + ship_length):
                            if(board[i][k]!=board[i][j]):
                                raise ShipError
                        ships[board[i][j]][ship_count]= [[i], list(range(j, j + ship_length)), ship_length]
                        # It passes the founded ship.
                        j += ship_length - 1
                    # If not, another try block checks for a vertically positioned ship.
                    except:
                        try:
                            for k in range(i + 1, i + ship_length):
                                if(board[k][j]!=board[i][j]):
                                    raise ShipError
                            ships[board[i][j]][ship_count] = [list(range(i, i + ship_length)), [j], ship_length]
```

```
                # It adds as many column numbers as it indicates how many rows this column
will be bypassed.
                    for l in range(ship_length):
                        ship_positions.append(j)
                except:
                    pass
            j += 1
        # It removes one of each different number before going on each new row.
        for number in range(len(board)):
            try:
                ship_positions.remove(number)
            except:
                pass
    for ship_type in ships:
        assert len(ships[ship_type]) == default_ship_counts(ship_type)
    return ships


def did_it_sink(ships, ship_type, health, ship):
    # It checks if the ship has sunk and if it has sunk it deletes the ship from the dict.
    if(health==0):
        ships[ship_type].pop(ship)
    # If there are no ships of that type left, it deletes that type from the dictionary.
    if(len(ships[ship_type])==0):
        ships.pop(ship_type)


def play(board, ships, coordinates):
    r = int(coordinates[0]) - 1
    c = alphabet.index(coordinates[1])
    assert 0<=r<10 and c<10
    square = board[r][c]
    if(square=="-"):
```

```
            board[r][c] = "O"
        elif(square=="O" or square=="X"):
            None
        else:
            board[r][c] = "X"
            try:
                # It finds which ship the hit ship is and reduces its health by one.
                for ship in ships[square]:
                    for horizontal in ships[square][ship][0]:
                        if(horizontal==r):
                            for vertical in ships[square][ship][1]:
                                if(vertical==c):
                                    ships[square][ship][2] -= 1
                                    did_it_sink(ships, square, ships[square][ship][2], ship)
                                    # Error given to exit the loop.
                                    raise RuntimeError
            # Exception for when the ship is found and gives a runtime error.
            except RuntimeError:
                pass

# Function used to create rows of hidden tables' of players'.
def create_row(board, i):
    row = "{:<2,d}".format(i + 1)
    for j in range(len(board[i])):
        if(board[i][j]=="O" or board[i][j]=="X"):
            row += "{} ".format(board[i][j])
        else:
            row += "{} ".format("-")
    row.rstrip()
    return row
```

```
# Function used to create hidden tables of players'.
def create_table(board1, board2):
    table = "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 i in range(len(board1)):
        table += create_row(board1, i) + "\t\t" + create_row(board2, i) + "\n"
    return table + "\n"


# Function that finds the number of remaining ships.
def ship_counter(ships, ship_type):
    try:
        return len(ships[ship_type])
    except KeyError:
        return 0


# Function that creates the ship's chart.
def ship_chart(ships, ship_type):
    ship_count = ship_counter(ships, ship_type)
    default_ship_count = default_ship_counts(ship_type)
    return ((default_ship_count - ship_count) * "X " + ship_count * "- ").rstrip()


# Function that creates the ship table.
def create_ship_table(ships1, ships2):
    c1 = ship_chart(ships1, "C")
    b1 = ship_chart(ships1, "B")
    d1 = ship_chart(ships1, "D")
    s1 = ship_chart(ships1, "S")
    p1 = ship_chart(ships1, "P")
    c2 = ship_chart(ships2, "C")
    b2 = ship_chart(ships2, "B")
```

```
    d2 = ship_chart(ships2, "D")
    s2 = ship_chart(ships2, "S")
    p2 = ship_chart(ships2, "P")
    table = """Carrier\t\t{}\t\t\tCarrier\t\t{}\nBattleship\t{}\t\t\tBattleship\t{}
Destroyer\t{}\t\t\tDestroyer\t{}\nSubmarine\t{}\t\t\tSubmarine\t{}
Patrol Boat\t{}\t\t\tPatrol Boat\t{}""".format(c1, c2, b1, b2, d1, d2, s1, s2, p1, p2)
    return table

# Function that creates the table of each round.
def create_output(board1, board2, ships1, ships2):
    table = create_table(board1, board2)
    ship_table = create_ship_table(ships1, ships2)
    return table + ship_table

try:
    output = ""

    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    board1 = create_board(sys.argv[1])
    board2 = create_board(sys.argv[2])

    input1 = read_in_files(sys.argv[3])
    input2 = read_in_files(sys.argv[4])

    ships1 = find_ships(board1)
    ships2 = find_ships(board2)

    base_table = create_output(board1, board2, ships1, ships2)
```

```
base_intro = "Player1's Move\n\nRound : 1\t\t\t\t\tGrid Size: 10x10\n\n"

output += "Battle of Ships Game\n\n" + base_intro + base_table + "\n\n"

player1_counter = 0
player2_counter = 0
round_counter = 1
turn = 1
# It shows who the winner is. It is 1 for player1, -1 for player2, 0 for draw.
win = 0

while True:
    if(turn==1):
        # If an error occurs in one of the turns, it uses if statements to identify the cause of the error and append it to the output.
        # If an error occurs, it stays in the same round until it is played without error.
        command = input1[player1_counter]
        try:
            coordinates = command.split(",")
            output += "Enter your move: {}\n".format(command)
            play(board2, ships2, coordinates)
            table = create_output(board1, board2, ships1, ships2)
            intro = "\nPlayer{:d}'s Move\n\nRound : {:d}\t\t\t\t\tGrid Size: 10x10\n\n".format(2, round_counter)
            output += intro + table + "\n\n"
            turn = 2
            if(len(ships2)==0):
                win += 1
        except IndexError:
            if(len(coordinates)==1):
                if(coordinates[0]==""):
```

```
                message = "This command doesn't have any arguments"
            else:
                message = "This command has just one argument"
        else:
            if(coordinates[0]==""):
                if(coordinates[1]==""):
                    message = "This command's arguments are empty"
                else:
                    message = "This command's first argument is empty"
            else:
                message = "This command's second argument is empty"
        output += "IndexError: {}.\n".format(message)
        round_counter -= 1
    except ValueError:
        try:
            int(coordinates[0])
        except:
            try:
                alphabet.index(coordinates[1])
            except:
                message = "First argument is a non-numeric value and second argument isn't
a letter"
            else:
                message = "First argument is a non-numeric value"
        else:
            try:
                alphabet.index(coordinates[1])
            except:
                message = "Second argument isn't a letter"
        output += "ValueError: {}.\n".format(message)
        round_counter -= 1
```

```
        except AssertionError:
            output += "AssertionError: Invalid Operation.\n"
            round_counter -= 1
        finally:
            player1_counter += 1
    if(turn==2):
        command = input2[player2_counter]
        try:
            coordinates = command.split(",")
            output += "Enter your move: {}\n".format(command)
            play(board1, ships1, coordinates)
            table = create_output(board1, board2, ships1, ships2)
            if(len(ships1)==0):
                win -= 1
                break
            intro = "\nPlayer{:d}'s Move\n\nRound : {:d}\t\t\t\t\tGrid Size:
10x10\n\n".format(1, round_counter + 1)
            output += intro + table + "\n\n"
            turn = 1
        except IndexError:
            if(len(coordinates)==1):
                if(coordinates[0]==""):
                    message = "This command doesn't have any arguments"
                else:
                    message = "This command has just one argument"
            else:
                if(coordinates[0]==""):
                    if(coordinates[1]==""):
                        message = "This command's arguments are empty"
                    else:
                        message = "This command's first argument is empty"
```

```
                else:
                    message = "This command's second argument is empty"
                output += "IndexError: {}.\n".format(message)
                round_counter -= 1
            except ValueError:
                try:
                    int(coordinates[0])
                except:
                    try:
                        alphabet.index(coordinates[1])
                    except:
                        message = "First argument is a non-numeric value and second argument isn't
a letter"
                    else:
                        message = "First argument is a non-numeric value"
                else:
                    try:
                        alphabet.index(coordinates[1])
                    except:
                        message = "Second argument isn't a letter"
                output += "ValueError: {}.\n".format(message)
                round_counter -= 1
            except AssertionError:
                output += "AssertionError: Invalid Operation.\n"
                round_counter -= 1
            finally:
                player2_counter += 1
        round_counter += 1


    if(win==1):
        winner = "Player1 Wins!"
```

```
    elif(win==-1):
        winner = "Player2 Wins!"
    else:
        winner = "Player1 Wins!, Player2 Wins!, It is a Draw!"


    final = "\n{}\n\nFinal Information\n\n{}".format(winner, table)
    output += final
except IOError:
    correct_inputs = ["Assignment4.py", "Player1.txt", "Player2.txt", "Player1.in",
"Player2.in"]
    incorrect_inputs = [correct_inputs[i] for i in range(1,5) if sys.argv[i]!=correct_inputs[i]]
    output += "IOError: input file(s) {} is/are not reachable.".format(",
".join(incorrect_inputs))
except:
    output += "kaBOOM: run for your life!"


write_output("Battleship.out", output)
print(output)
```

While creating the program, I paid attention to its flexibility. Much of the work that the program does is severely fragmented and divided into functions. In this way, with minor changes to the game, other programmers can make different games according to their wishes. Almost all functions of the program determine the function inputs according to the state of the board given with the input, not the constant numbers. In this way, a game with different rules can be implemented with very little change, or a game can be written whose rules change as it is played.

USER CATALOGUE

The program uses *Player1.txt, Player2.txt*, where player boards are given, and *Player1.in, Player2.in* files, where player moves are given as inputs. Please make sure these files are in the same file as the program.

Ships can be placed horizontally or vertically.

The ship length is 5 for a carrier, 4 for a battleship, 3 for a submarine and a destroyer, and 2 for a patrol boat.

Number of ships 1 for carrier, submarine, destroyer; 2 for battleship and 4 for patrol boat.

The board length is 10*10.

When filling the board, you must fill it according to the specifications given above.

Ranges on the board will be displayed with *";"*. You must write the initials of the ships in the spaces where you will place the ships.

As you fill out your in files, you must specify each move as *"[row number],[column's letter];"*.

You can run the program via console.

Navigate to the folder where the program and files are located in the console.

Type *"Assignment4.py Player1.txt Player2.txt Player1.in Player2.in"* into the console.

The program will print the output both to the console and to the output file named *Battleship.out*.

PROGRAM RESTRICTIONS

If you give the board different from the specified length, the program will give an error.

If the game doesn't end with the in file you have given, the program will give an error.

If the number of ships on the board you given is different from the specified one, the program will throw an error.

If you enter anything other than ship letters on the board, the program will give an error.

GRADING TABLE

Evaluation	Points	Evaluate Yourself / Guess Grading
Using Explanatory Comments	5	5
Efficiency (avoiding unnecessary actions)	5	5
Function Usage	15	15
Correctness, File I/O	30	30
Exceptions	20	20
Report	20	20