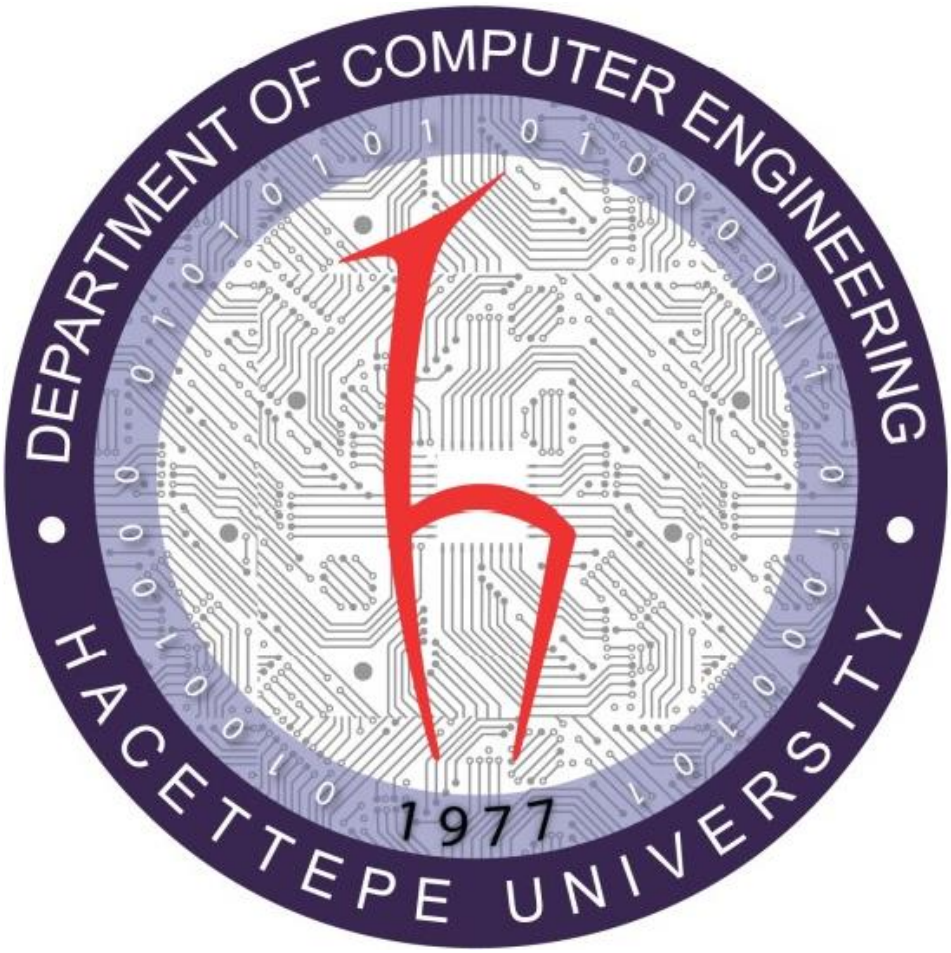
Hacettepe University
Department of Computer Science

BBM-101 Assignment 4 Report

Gökdeniz Şimşek – 2210356067

3.01.2023



ANALYSIS

In this section, I discuss the problem of the assignment and explain the aims of the project.

In this assignment, we need to create a Battle of Ships game that takes data from different files and proceeds according to that data. To go into detail, we need to get some information from six different files belonging to two different players. This data includes information on the players' ship locations, specific locations for some of the players' ships (such as the Patrol Boat and Battleship), and the location of the player's moves. Afterward, a code block should be created that will use this information correctly. Game boards must be created first for this game. These should be a board showing information about the location of the player's ships and a hidden board. These boards, which are created according to the entered files, are reshaped according to the moves of the players in each round. In addition, after the moves of both players, the information of the hidden board and the hit ships should be given as output. These processes continue until all ship positions determined by a player are hit by the other player. When the game is over, the last state of the hidden boards and ships is given as an output, along with the name of the winning player. The most important part of this assignment is debugging. Incorrect or unresolved information entered by the players causes an error. According to the type of these errors, the output should be given to the players. If there is an error other than IndexError, ValueError, or AssertionError encountered, it should give "kaBOOM" output and the program should be terminated.

DESIGN

Find a Solution

Firstly, the problem should be considered in general, and necessary solutions should be sought. The fundamental functions and code blocks required for the solution should be created by evaluating all possible errors. When the input files are examined, the information in the given data should be parsed and made usable by the necessary codes and functions. Special functions should be written for code blocks that are used repeatedly, such as ship information and hidden board information. Debugging should be done where necessary.

Getting Input from Files

In order to get input from the files, the files should be opened with the "r" key, and the files should be evaluated separately for each line. In this case, the "readline" or "readlines" functions can be used. After getting information about the locations of the ships from Player1.txt and Player2.txt files, the files OptionalPlayer1.txt and OptionalPlayer2.txt should be read for the confusing "Battleships" and "Patrol Boats". The game is ready to play after reading these four files. Finally, the Player1.in and Player2.in files should be read, and actions should be taken according to the moves of the players.

Locating Ships and Creating Boards

Files Player1.txt and Player2.txt must be read first to locate ships. There are nine ";" in each line of the Player1.txt and Player2.txt files. The files should contain a total of 10 lines. Each row represents a row on the player board. And each letter to the right and left of the ";" represents the position of the ship associated with that letter on the Player's Board. After getting information about the locations of the ships from the Player1.txt and Player2.txt files, the files OptionalPlayer1.txt and OptionalPlayer2.txt should be read for the confusing "Battleships" and "Patrol Boats". Each line in these files represents one of these ships. The lines of this file include the name of the ship, the starting point of the ship, and the direction in which it continues. The exact positions of these six ships in total are determined precisely by reading these files.

The Player Board is the board on which the player must show the location of his ships. On this board, the initials of the ships are used for the places where the ships are located, and the "-" sign is used for the spaces. The opposing player uses the Player's Hidden Board to see where they shoot. In the beginning, each position must have a "-" sign. If the player fired at the opponent's ship's position, the "-" sign is replaced with an "X". If a shot is fired to a point that is not a ship, the "-" sign is replaced with an "O".

Making the Moves of the Players

In order to use the moves of the players, the files Player1.in and Player2.in must be read first. Then each move in the files separated by ";" must be executed sequentially. However, a move after the first player's file must be taken from the second player's file. It should continue like this throughout the entire game. (Player 1 makes one move, and Player 2 makes the other. Then return to Player 1.) Debugging should also be done while making these moves. If the moves entered by the players are incorrect, it should give output according to the type of error. Then, instead of that move, the next move of the player should be taken from the Player.in file. In this way, it is ensured that each player makes exactly one move per turn.

Exception Handling

First of all, the program should check if the files exist in the filenames entered by the players. If there are errors in the files, it should give an IOError and give output indicating which files are wrong.

Then, if the data obtained from the Player.in and Player2.in files is incorrect while the player's moves are being made, the necessary error output should be given. In order to determine the type of error, the data taken from the file should be examined. When the left or right side of the "," in each move data separated by ";" is blank, an IndexError is returned. For example, ",A;", "A;", ",;", ";", "1,;", "1;" are some of these situations. ValueError is received when there is incorrect data to the right or left of the "," sign. Normally, the line number of the move to be made should be on the left of the "," sign, and the letter indicating the column of the move should be on the right. For example, "A,1;", "1,1;", "A,A;", "5,E10,G;" are some of these situations. When a player's move does not match any location on the Player Board, an AssertionError is thrown. For example: "11,A;", "5,K;" are some of these situations.

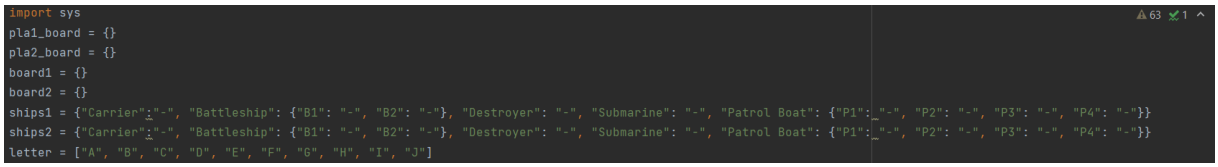
If there is any error except IOError, IndexError, ValueError, and AssertionError, the program must give “kaBOOM” output and shut down itself.

Giving Final Information

The game must be over if one player shoots at all of the ship positions determined by the other player. This means that all of a player's ships are sunk. If the first player makes the last shot in this situation, the second player must be given another chance to equalize the number of moves. If the other player also hits all the ships with his last shot, a printout must be given stating that they are tied. Final Information should also print out a board showing the shooting locations and the locations of the ships that were not hit, as well as a chart showing the ships that were hit.

Programmer Catalog

Importing sys Module and Dictionaries



```
import sys
pla1_board = {}
pla2_board = {}
board1 = {}
board2 = {}
ships1 = {"Carrier":"-", "Battleship": {"B1": "-", "B2": "-"}, "Destroyer": "-", "Submarine": "-", "Patrol Boat": {"P1":"-", "P2": "-", "P3": "-", "P4": "-"}}
ships2 = {"Carrier":"-", "Battleship": {"B1": "-", "B2": "-"}, "Destroyer": "-", "Submarine": "-", "Patrol Boat": {"P1":"-", "P2": "-", "P3": "-", "P4": "-"}}
letter = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]
```

We imported the Sys module into our program because the names of the files we will receive as input will be the arguments given by the user. Also here, we have created some dictionaries and letter list that we will use in functions or code blocks in the future.

Checking the Existence of Files

```
file_1 = sys.argv[1]
file_2 = sys.argv[2]
f_in_1 = sys.argv[3]
f_in_2 = sys.argv[4]
try:
    with open(file_1, "r", encoding="utf-8") as ply1:
        ply1.close()
except IOError:
    try:
        with open(file_2, "r", encoding="utf-8") as ply1:
            ply1.close()
    except IOError:
        try:
            with open(f_in_1, "r", encoding="utf-8") as pl1in:
                pl1in.close()
        except IOError:
            try:
                with open(f_in_2, "r", encoding="utf-8") as pl2in:
                    pl2in.close()
                print(f"IOError: input files {file_1}, {file_2}, {f_in_1} are not reachable.")
            except IOError:
                print(f"IOError: input files {file_1}, {file_2}, {f_in_1}, {f_in_2} are not reachable.")
        else:
            try:
                with open(f_in_2, "r", encoding="utf-8") as pl2in:
                    pl2in.close()
                print(f"IOError: input files {file_1}, {file_2} are not reachable.")
            except IOError:
                print(f"IOError: input files {file_1}, {file_2}, {f_in_2} are not reachable.")
```

```
    else:
        try:
            with open(f_in_1, "r", encoding="utf-8") as pl1in:
                pl1in.close()
        except IOError:
            try:
                with open(f_in_2, "r", encoding="utf-8") as pl2in:
                    pl2in.close()
                print(f"IOError: input files {file_1}, {f_in_1} are not reachable.")
            except IOError:
                print(f"IOError: input files {file_1}, {f_in_1}, {f_in_2} are not reachable.")
        else:
            try:
                with open(f_in_2, "r", encoding="utf-8") as pl2in:
                    pl2in.close()
                print(f"IOError: input files {file_1} is not reachable.")
            except IOError:
                print(f"IOError: input files {file_1}, {f_in_2} are not reachable.")
```

```
else:
    try:
        with open(file_2, "r", encoding="utf-8") as ply1:
            ply1.close()
    except IOError:
        try:
            with open(f_in_1, "r", encoding="utf-8") as pl1in:
                pl1in.close()
        except IOError:
            try:
                with open(f_in_2, "r", encoding="utf-8") as pl2in:
                    pl2in.close()
                print(f"IOError: input files {file_2}, {f_in_1} are not reachable.")
            except IOError:
                print(f"IOError: input files {file_2}, {f_in_1}, {f_in_2} are not reachable.")
        else:
            try:
                with open(f_in_2, "r", encoding="utf-8") as pl2in:
                    pl2in.close()
                print(f"IOError: input files {file_2} is not reachable.")
            except IOError:
                print(f"IOError: input files {file_2}, {f_in_2} are not reachable.")
    else:
        try:
            with open(f_in_1, "r", encoding="utf-8") as pl1in:
                pl1in.close()
        except IOError:
            try:
                with open(f_in_2, "r", encoding="utf-8") as pl2in:
                    pl2in.close()
                print(f"IOError: input files {f_in_1} is not reachable.")
            except IOError:
                print(f"IOError: input files {f_in_1}, {f_in_2} are not reachable.")
        else:
            try:
                with open(f_in_2, "r", encoding="utf-8") as pl2in:
                    pl2in.close()
            except IOError:
                print(f"IOError: input files {f_in_2} is not reachable.")
            else:
                with open("Battleship.out", "w", encoding="utf-8") as bat_out:
                    bat_out.close()
                with open(file_1, "r", encoding="utf-8") as ply1:
                    for nums1 in range(1, 11):
                        line1 = ply1.readline()
                        if line1 != "":
```

I have created a code block that opens the files in order and progresses according to the errors. In case of an error, the program terminates without running the remaining codes. The only output given is the names of the incorrectly entered files. If there is no error, the innermost else block runs, and the main part of the program starts to run.

Locating Ships and Creating Boards

```
with open(file_1, "r", encoding="utf-8") as ply1:
    for nums1 in range(1, 11):
        line1 = ply1.readline()
        if line1 != "":
            line1 = line1.strip().split(";")
            for j in range(len(line1)):
                if len(line1[j]) > 1:
                    raise Exception
                elif line1[j] == "":
                    line1[j] = "-"
            for i in range(len(letter)):
                pla1_board[str(nums1) + letter[i]] = line1[i]
    with open("OptionalPlayer1.txt", "r", encoding="utf-8") as opt1:
        for h in range(6):
            line_1 = opt1.readline().strip().split(":")
            ship = line_1[0]
            line_1 = line_1[1].split(";")
            point = line_1[0]
            direction = line_1[1]
            def_ships(ship, point, direction, pla1_board)
```

```
with open(file_2, "r", encoding="utf-8") as ply1:
    for nums2 in range(1, 11):
        line2 = ply1.readline()
        if line2 != "":
            line2 = line2.strip().split(";")
            for j in range(len(line2)):
                if len(line2[j]) > 1:
                    raise Exception
                elif line2[j] == "":
                    line2[j] = "-"
            for i in range(len(letter)):
                pla2_board[str(nums2) + letter[i]] = line2[i]
    with open("OptionalPlayer2.txt", "r", encoding="utf-8") as opt2:
        for h in range(6):
            line_2 = opt2.readline().strip().split(":")
            ship = line_2[0]
            line_2 = line_2[1].split(";")
            point = line_2[0]
            direction = line_2[1]
            def_ships(ship, point, direction, pla2_board)
```

Since the files containing the locations of the player ships are taken as arguments, I assigned the variables file_1 (Player1.txt) and file_2 (Player2.txt) and used these variables when opening the files. Then I split each line by the split function from the "," sign. Then I checked the elements of the resulting list one by one. If the element is a space, I replaced it with a "-" sign. Then I assign elements to positions on the board by using a for loop to create the Player Board using a dictionary. (Letters for ships, "-" for spaces.) I raised the Exception if there is an error.

The OptionalPlayer.txt files, which indicate the locations of the Patrol Boat and Battleship ships, are read directly without any arguments. Again, I examined each line separately to be able to read these files. I separated the line from the ":" with the split function. I assigned the first item of the resulting list to the variable named "ship". I separated the second item of the list from the ";" sign by split function. I equalized the first item of this resulting list to the "point" variable and the second item to the "direction" variable. Finally, I passed these variables inside the def_ship function.

The def_ship Function

```
def def_ships(ship,point,direction,pla_board):
    point_list = point.split(",")
    if ship[0] == "B":
        if direction == "right":
            for i in letter[letter.index(point_list[1]):letter.index(point_list[1]) + 4]:
                pla_board[point_list[0]+i] = ship
        elif direction == "down":
            for j in range(4):
                pla_board[str(int(point_list[0])+j)+point_list[1]] = ship
    elif ship[0] == "P":
        if direction == "right":
            for i in letter[letter.index(point_list[1]):letter.index(point_list[1]) + 2]:
                pla_board[point_list[0]+i] = ship
        elif direction == "down":
            for j in range(2):
                pla_board[str(int(point_list[0])+j)+point_list[1]] = ship
```

This function works with the previously created ship, point, and direction variables and the pla_board variable that specifies which player it belongs to. In order to find out from which point the given ship started, I separated the "point" variable from the "," by the split function and equated the resulting list to the "point_list". Then, I started to change the dictionary value of the points on the Player Board with the ship name, starting from the starting point, depending on the type of ship. I swapped 4 units for Battleships and 2 units for Patrol Boats in the direction of the given "direction" variable.

Making the Moves of the Players

```
with open(f_in_1, "r", encoding="utf-8") as p1in:
    p1in_list = p1in.readline().split(";")
    p1in_list = p1in_list[:-1]
with open(f_in_2,"r", encoding="utf-8") as p2in:
    p2in_list = p2in.readline().split(";")
    p2in_list = p2in_list[:-1]
for z in list(pla1_board.keys()):
    board1[z] = "-"
for x in list(pla2_board.keys()):
    board2[x] = "-"
values1 = list(pla1_board.values())
values2 = list(pla2_board.values())
pla1_tot = "B1" in values1 or "B2" in values1 or "C" in values1 or "D" in values1 or "P1" in values1 or "P2" in values1 or "P3" in values1 or "P4" in values1 or "S" in values1
pla2_tot = "B1" in values2 or "B2" in values2 or "C" in values2 or "D" in values2 or "P1" in values2 or "P2" in values2 or "P3" in values2 or "P4" in values2 or "S" in values2
```

Since we get the player moves from the file, we first created a list of moves by separating the line in the file from the "," sign with the split function. Also in this part, I created the hidden board that we will use for shots. And here I created pla1_tot and pla2_tot, which are the control variables of the while loop that I will use for the shots.

```
while (pla1_tot and pla2_tot) and k < len(p1in_list):
    try:
        output(f"Player1's Move\n\nRound : {n}\t\t\t\t\tGrid Size: 10x10\n\n")
        output(board_print(board1, board2))
        output(ship_print())
        pla1_cho = p1in_list[k]
        point_check = pla1_cho.split(",")
        if len(point_check[0]) < 1 or len(point_check[1]) < 1:
            raise IndexError
        str(point_check[1])
        int(point_check[0])
        if len(point_check) > 2 or len(point_check[1]) > 1:
            raise ValueError
        output("\n\nEnter your move: " + pla1_cho + "\n\n")
        pla1_cho = pla1_cho.split(",")
        point = pla1_cho[0] + pla1_cho[1]
        assert point in pla1_board.keys()
```

```
except IndexError:
    output("\n\nIndexError: Invalid move made.\n\n")
    k += 1
    continue
except ValueError:
    output("\n\nValueError: Invalid move made.\n\n")
    k += 1
    continue
except AssertionError:
    output("\n\nAssertionError: Invalid Operation.\n\n")
    k += 1
    continue
else:
    if pla2_board[point] != "-":
        board2[point] = "X"
        pla2_board[point] = "X"
    else:
        board2[point] = "O"
        pla2_board[point] = "O"
    check_ship(pla2_board, ships2)
except IndexError:
    output("\n\nIndexError: Invalid move made.\n\n")
    k += 1
    continue
except ValueError:
    output("\n\nValueError: Invalid move made.\n\n")
    k += 1
    continue
except AssertionError:
    output("\n\nAssertionError: Invalid Operation.\n\n")
    k += 1
    continue
else:
    if pla2_board[point] != "-":
        board2[point] = "X"
        pla2_board[point] = "X"
    else:
        board2[point] = "O"
        pla2_board[point] = "O"
    check_ship(pla2_board, ships2)
    while pla2_tot and j < len(p2in_list):
        try:
            output(f"Player2's Move\n\nRound : {n}\t\t\t\t\tGrid Size: 10x10\n\n")
            output(board_print(board1, board2))
            output(ship_print())
            pla2_cho = p2in_list[j]
            point_check = pla2_cho.split(",")
            if len(point_check[0]) < 1 or len(point_check[1]) < 1:
                raise IndexError
            str(point_check[1])
            int(point_check[0])
```

```
            if len(point_check) > 2 or len(point_check[1]) > 1:
                raise ValueError
            output("\n\nEnter your move: " + pla2_cho + "\n\n")
            pla2_cho = pla2_cho.split(",")
            point = pla2_cho[0] + pla2_cho[1]
            assert point in pla2_board
        except IndexError:
            output("\n\nIndexError: Invalid move made.\n\n")
            j += 1
            continue
        except ValueError:
            output("\n\nValueError: Invalid move made.\n\n")
            j += 1
            continue
        except AssertionError:
            output("\n\nAssertionError: Invalid Operation.\n\n")
            j += 1
            continue
        else:
            if pla1_board[point]!= "-":
                board1[point] = "X"
                pla1_board[point] = "X"
                check_ship(pla1_board, ships1)
                break
            else:
                board1[point] = "0"
                pla1_board[point] = "0"
                check_ship(pla1_board, ships1)
                break
```

```
    j += 1
    k += 1
    n += 1
    values1 = list(pla1_board.values())
    values2 = list(pla2_board.values())
    pla1_tot = "B1" in values1 or "B2" in values1 or "C" in values1 or "D" in values1 or "P1" in values1 or "P2" in values1 or "P3" in values1 or "P4" in values1 or "S" in values1
    pla2_tot = "B1" in values2 or "B2" in values2 or "C" in values2 or "D" in values2 or "P1" in values2 or "P2" in values2 or "P3" in values2 or "P4" in values2 or "S" in values2
```

While shooting, I used two nested while loops to get moves from two players in turn. The condition for while loops are there is a ship is not hit and the shots entered by the user are not unfinished. After printing the information that should be given at the beginning of the round, the debugging process starts at the point where the user enters. I created the "point_check" variable for this. The code continues if the program does not give an error after making these necessary operations to the variable. It is created by the point split function and string addition operations. Then the assertion error is checked. If an IndexError, ValueError, or AssertionError is received, the same operations are repeated with the next move in the player's move list, and so on until an error-free move is made. If there is no error, the position of the found point on the Player Board is checked. If there is a ship at that point, that point is replaced with "X" on both the Player Board and the Player Hidden Board. If there is no ship at that point, it is replaced with "O". Before proceeding to the second while loop, the "check_ship" function is used to check whether the ships are sunk or not. Then, the second player's while loop starts. The same operations are performed as in the first loop. Only in the last step, if a point on the boards is replaced by an "X" or "O", the second while loop is terminated with the break command after the check_ship function runs. At the end of the outer while loop, the j, k, and l variables are incremented by 1, and the conditions of the while loops are checked.

The check_ship Function

```
def check_ship(playerboard, ship):
    if not "B1" in playerboard.values():
        ship["Battleship"]["B1"] = "X"
    if not "B2" in playerboard.values():
        ship["Battleship"]["B2"] = "X"
    if not "P1" in playerboard.values():
        ship["Patrol Boat"]["P1"] = "X"
    if not "P2" in playerboard.values():
        ship["Patrol Boat"]["P2"] = "X"
    if not "P3" in playerboard.values():
        ship["Patrol Boat"]["P3"] = "X"
    if not "P4" in playerboard.values():
        ship["Patrol Boat"]["P4"] = "X"
    if not "C" in playerboard.values():
        ship["Carrier"] = "X"
    if not "S" in playerboard.values():
        ship["Submarine"] = "X"
    if not "D" in playerboard.values():
        ship["Destroyer"] = "X"
```

This function checks the values on the Player Board to show which ships have completely sunk. It takes two different arguments. These arguments only specify which player will control the Player Board and ships. If the function cannot find any point belonging to a ship on the given Player Board, it converts the value of that ship from the "-" sign to "X" in that player's ships dictionary.

The board_print and ship_print Functions

```
def board_print(board_1, board_2):
    board_out = "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(1, 11):
        if i < 10:
            board_out += f"{i} "
            for j in letter:
                board_out += f"{board_1[str(i) + j]} "
            board_out = board_out[:-1]
            board_out += f"\t\t{i} "
            for j in letter:
                board_out += f"{board_2[str(i) + j]} "
            board_out = board_out[:-1]
            board_out += "\n"
        else:
            board_out += f"{i}"
            for j in letter:
                board_out += f"{board_1[str(i)+j]} "
            board_out = board_out[:-1]
            board_out += f"\t\t{i}"
            for j in letter:
                board_out += f"{board_2[str(i) + j]} "
            board_out = board_out[:-1]
            board_out += "\n\n"
    return board_out
```

The board_print function allows us to provide information about the game board and the number of rounds at the end of each shot. The values on the Player Board change after each shot. This function is also arranged and printed according to the values in the Player Boards.

```python
def ship_print():
    ship_out = ""
    for i in ships1:
        if i == "Battleship":
            ship_out += f"{i}\t"
            for j in sorted(list(ships1[i].values()), reverse=True):
                ship_out += f"{j} "
            ship_out = ship_out[:-1]
            ship_out += f"\t\t\t\t{i}\t"
            for j in sorted(list(ships2[i].values()), reverse=True):
                ship_out += f"{j} "
            ship_out = ship_out[:-1]
            ship_out += "\n"
        elif i == "Patrol Boat":
            ship_out += f"{i}\t"
            for j in sorted(list(ships1[i].values()), reverse=True):
                ship_out += f"{j} "
            ship_out = ship_out[:-1]
            ship_out += f"\t\t\t{i}\t"
            for j in sorted(list(ships2[i].values()), reverse=True):
                ship_out += f"{j} "
            ship_out = ship_out[:-1]
        elif i == "Carrier":
            ship_out += f"{i}\t\t{ships1[i]}\t\t\t\t{i}\t\t{ships2[i]}"
            ship_out += "\n"
        else:
            ship_out += f"{i}\t{ships1[i]}\t\t\t\t{i}\t{ships2[i]}"
            ship_out += "\n"
    return ship_out
```

The board_print function allows us to provide information about the which ships are hit at the end of each shot. This function gets its data from dictionaries named ships1 and ships2. According to the check_ship function, the values in this dictionary change. If a ship is hit completely, that ship should be shown with an "X" on the ship's hit table. This function allows printing this table at the end of each round.

```
Player1's Move

Round : 82                                        Grid Size: 10x10

Player1's Hidden Board          Player2's Hidden Board
   A B C D E F G H I J             A B C D E F G H I J
1  0 0 0 0 - 0 X - 0 0          1  - - 0 0 0 0 0 - 0 -
2  - 0 0 0 - 0 X 0 0 0          2  - 0 X X X X X 0 0 X
3  - - - 0 X 0 X X X 0          3  - 0 0 0 0 0 0 0 0 X
4  0 X - 0 X 0 X 0 - 0          4  X 0 X X 0 0 0 0 0 0
5  0 0 0 0 X 0 X 0 - 0          5  - - 0 - 0 0 X X - X
6  - X X X X - 0 0 0 0          6  0 0 0 0 0 - - 0 0 0 -
7  0 0 0 0 0 0 X X X 0 0        7  0 X - 0 - X 0 0 0 0
8  0 0 - 0 0 0 0 - 0 X          8  0 - - 0 0 0 0 X 0 0
9  - - 0 - X X 0 0 0 X          9  - X 0 X X 0 0 X 0 -
100 X X 0 0 - 0 - 0 X           100 X 0 0 0 0 0 0 0 0 0

Carrier          X                                Carrier          X
Battleship       X -                              Battleship       - -
Destroyer        X                                Destroyer        -
Submarine        X                                Submarine        -
Patrol Boat      X X X -                  Patrol Boat      X X X -
```

Editing Values on the Player Boards

```
for keys in pla1_board.keys():
    if pla1_board[keys] in ["P1","P2","P3","P4"]:
        pla1_board[keys] = "P"
    if pla1_board[keys] in ["B1","B2"]:
        pla1_board[keys] = "B"
for keys in pla2_board.keys():
    if pla2_board[keys] in ["P1","P2","P3","P4"]:
        pla2_board[keys] = "P"
    if pla2_board[keys] in ["B1","B2"]:
        pla2_board[keys] = "B"
```

In this code block, it is checked whether there are Patrol Boat and Battleship ships on the Player Board at the end of the shots. If any, the value of that point in the dictionary is replaced with the initial letter of the ship only. (B instead of B1, B2 and P instead of P1, P2, P3, P4)

Giving Final Information

```
values1 = list(pla1_board.values())
values2 = list(pla2_board.values())
pla1_tot = "B1" in values1 or "B2" in values1 or "C" in values1 or "D" in values1 or "P1" in values1 or "P2" in values1 or "P3" in values1 or "P4" in values1 or "S" in values1
pla2_tot = "B1" in values2 or "B2" in values2 or "C" in values2 or "D" in values2 or "P1" in values2 or "P2" in values2 or "P3" in values2 or "P4" in values2 or "S" in values2
if not pla1_tot and not pla2_tot:
    output("Player1 Wins!\nPlayer2 Wins!\nIt is a Draw!\n\nFinal Information\n\n")
    output(board_print(pla1_board, pla2_board))
    output(ship_print())
elif not pla1_tot:
    output("Player2 Wins!\n\nFinal Information\n\n")
    output(board_print(pla1_board, pla2_board))
    output(ship_print())
elif not pla2_tot:
    output("Player1 Wins!\n\nFinal Information\n\n")
    output(board_print(pla1_board, pla2_board))
    output(ship_print())
```

In the last part of the program, it is checked again to see which player has any not-shot ships. At least one of the players must have hit all the ships. If both have hit all ships, the output is given that it is draw. If only one of the players hits all of the ships, that player is declared the winner, and the output will be given accordingly. Then, the final states of the Player Boards and the hit ship charts will be given as an output.

Time Spent

For analyzing: 1 and half hours

For designing: 2 and half hours

For implementing: 6 and half hours

For testing: 1 hour

For reporting: 3 hours

User Catalog

Half of the program consists of functions, and the other half consists of code blocks. So there is not much that the user can change. However, when running the code in the terminal, four arguments must be entered, all of which are file names. The first two of these are Players.txt files, which specify the location of the ships, contain nine ";"s on each line, and should consist of 10 lines. The last two are Players.in files, where each move is separated by a ";" and must contain the player's moves. Each element in these files must be in the form of a "number, letter" separated by ";".

```
> python3 Assignment4.py "Player1.txt" "Player2.txt" "Player1.in" "Player2.in"
```

The command to be entered into the terminal should be as above.

Evaluation	Points	Evaluate Yourself / Guess Grading
Indented and Readable Codes	5	5
Using Meaningful Naming	5	5
Using Explanatory Comments	5	3
Efficiency (avoiding unnecessary actions)	5	5
Function Usage	25	25
Correctness	35	35
Report	20	20
There are several negative evaluations	...	...