COVER

Course Name: BBM 103

Assignment Number: 4

Student Number: 2220356080

Name Surname: Bora Aslan

Delivery Date: 27.11.2022

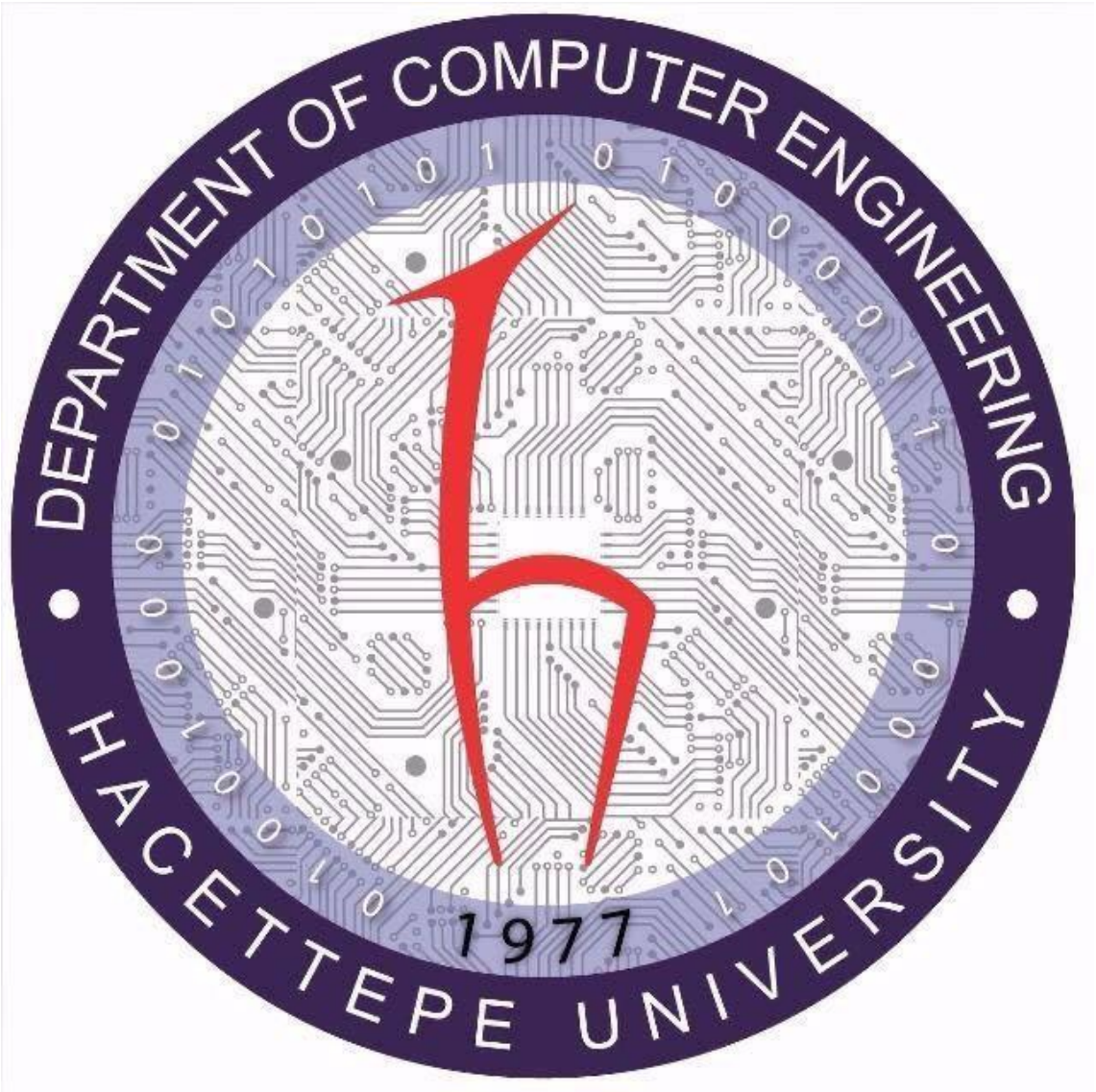


TABLE OF CONTENTS:

Grading Table

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	34
Report	20	18

ANALYSIS

The problem in this assignment is to read moves, operands, and optional inputs from different files. Which could be done by builtin python functions and recursive functions. Later on, we should check for errors, and handle exceptions. Using try-except blocks is the best way to accomplish these. Then we should combine the data, and create a list of ships and a matrix of the board to show which ship is in which square and whether it is floating or sunk. Loops and a recursive function can handle these things.FinallytThe program should also print each move and the board twice in every round to the terminal and a text file. After getting the backend these problems are child's toy. By using print and file.write these can be handled.

DESIGN

-FILE I/O

This function takes file names entered in the terminal and checks if each file exists using a for loop and the OS library. If one or more files do not exist, the code is stopped and an error message is printed.

The function then opens the given text files, splits them by new lines, and assigns each line to variables. This is done in a try-except block to prevent all kinds of errors. If an error occurs in this part, the code is stopped.

```
file_names = sys.argv[1:]
# takes names from terminal
existo, noexisto = [], []
f = open("Battleship.out", "w", encoding="utf8")
# Tries to open each file and raises error!
try:
    for file_name in file_names:
        if os.path.exists(file_name) is False:
            noexisto.append(file_name)
    if noexisto:
        raise IOError("IOError: input file(s) {} is/are not
reachable.".format(", ".join(noexisto)))

except IOError as msg:
    print(msg)
    a = "{}".format(msg)
    f.write(a)
    quit()
except Exception:
    print("kaBOOM: run for your life!")
    f.write("kaBOOM: run for your life!\n")
    quit()
# for errors if the files are corrupted

try:
    player1txt = open("Player1.txt", "r",
encoding="utf8").read().split("\n")
    player2txt = open("Player2.txt", "r",
encoding="utf8").read().split("\n")
    optplayer1txt = open("OptionalPlayer1.txt", "r",
encoding="utf8").read().split("\n")
    optplayer2txt = open("OptionalPlayer2.txt", "r",
encoding="utf8").read().split("\n")
    player1in = open("Player1.in", "r", encoding="utf8").read()
    player2in = open("Player2.in", "r", encoding="utf8").read()
except Exception as m:
    print(m, "Try again after checking files, game over!")

    f.write(m)
    quit()
    f.write(" Try again after checking files, game over!")
```

```
    quit()
```

Takes file names entered to terminal checks if each exists with for loop using OS library if one or more does not exist code is stopped and an error message is printed.

Later on, opens the given named text files splits them from new lines, and assigns each to variables

This is done in try and bare except block to prevent every kind of error if an error occurs in this part code is stopped!

```
# reads a player's input txts and creates a list to store only ships
positions
def txtreader(playertxt, optplayer, lsd):
    shipC, shipD, ship5, empty = ["C1"], ["D1"], ["51"], []
    for number, line in enumerate(playertxt):
        splitted = line.split(";")
        for letter, item in enumerate(splitted):
            place = [str(number + 1), converter(letter)]
            if item == "C":
                shipC.append(place)
            elif item == "D":
                shipD.append(place)
            elif item == "5":
                ship5.append(place)
            elif item == "":
                empty.append(place)
    lsd.append(shipD)
    lsd.append(ship5)
    lsd.append(shipC)
    for line in optplayer:
        line = line.replace(",", ";").replace(":", ";")
        parts = line.split(";")
        shipname, num, letter, direction = parts[:-1]

        if shipname[0] == "B":  # 4digits 2 ships
            temp = [shipname]
            for i in range(4):
                if direction == "down":
                    temp.append([str(int(num) + i), letter])
                else:
                    temp.append([num, converter(converter(letter) +
i)])

            lsd.append(temp)

        else:
            temp = [shipname]
            for i in range(2):
                if direction == "down":
                    temp.append([str(int(num) + i), letter])
                else:
                    temp.append([num, converter(converter(letter) +
i)])
            lsd.append(temp)
```

```
txtreader(player1txt, optplayer1txt, player1ships)
txtreader(player2txt, optplayer2txt, player2ships)
```

Function_converter(var):

This function works for only the first 10 letters of the alphabet by using an ordered list. It converts the nth letter in the alphabet to n by indexing and vice versa converts n to the nth letter in the alphabet by indexing.

Function_txtreader (playertxt, optplayer, lsd):

This function creates a list for ships C, D, S, and empty squares. It goes through each number and line in **enumerate(playertxt)** with a for loop. It then splits the line by **;** and goes through the letters and items in the split line with another for loop. It creates a **place** variable to append to the list of places more easily.

The function checks the value of the item and categorizes it, appending it to the appropriate ship's list. After this, it appends the ship lists to a temporary list.

The function then starts reading the optional texts. It replaces **,** with **;** and **:** with **;**, splits the line by **;**, and assigns the split items to variables. It checks the ship names and assigns them with a for loop. If a ship is 4 squares, it works 4 times, each time registering towards a specified direction.

Function_STARTER

This function prints the ordinary lines like the round count, hidden boards, and the **abcd** labels of the boards.

```
# function to print out general part of every move period
def starter(ct1):
    tmp1 = "Round : {}                    Grid Size:
10x10\n\n".format(
        ct1) + "Player1's Hidden Board      Player2's Hidden
Board\n" + "   A B C D E F G H I J\t      A B C D E F G H I J"
    tmp2 = "Round : {}                    Grid Size:
10x10\n\n".format(
        ct1) + "Player1's Hidden Board      Player2's Hidden
Board\n" + "   A B C D E F G H I J\t      A B C D E F G H I J\n"
    f.write(tmp2)
    print(tmp1)
```

Function_LISTER

This function prints the player's boards by using the player dictionaries' data. It uses string concatenation to create each line and then prints them all at the end of the loop.

```
# prints ship boards of each player by getting them from matrix and
adding to str
def lister(matrix1, matrix2):
    str1 = ""
    str2 = ""
    counter = 1
    for j, i in zip(matrix1, matrix2):
        if counter == 91:
            str1 += "10{}".format(matrix1[j])
            str2 += "10{}".format(matrix2[i])

        elif counter % 10 == 1:
            str1 += "{} {}".format(counter // 10 + 1, matrix1[j])
            str2 += "{} {}".format(counter // 10 + 1, matrix2[i])

        elif 10 > counter % 10 > 1:
            str1 += " {}".format(matrix1[j])
            str2 += " {}".format(matrix2[i])

        elif counter % 10 == 0:

            str1 += " {}".format(matrix1[j])
            str2 += " {}".format(matrix2[j])

            str1 += "\t    " + str2
            print(str1)
            f.write(str1 + "\n")

            str1 = ""
            str2 = ""
        counter += 1
    print("")
```

Function_SHIPS

```
# prints the ships sunk or float status
# by checking the values in players dictionaries
def ships():
    f.write("\n")
    count = 0
    for i in player1ships[2][1:]:
        if M1[tuple(i)] == "X":
            count += 1
    if count == 5:
        print("Carrier     X".ljust(28), end="")
        f.write("Carrier     X".ljust(28))
    else:
        f.write("Carrier     -".ljust(28))
        print("Carrier     -".ljust(28), end="")
    count = 0
    for i in player2ships[2][1:]:
        if M2[tuple(i)] == "X":
            count += 1
    if count == 5:
        print("Carrier     X")
        f.write("Carrier     X\n")
```

```
    else:
        print("Carrier     -")
        f.write("Carrier     -\n")

    count1, count2 = 0, 0
    for i in player1ships[3][1:]:
        if M1[tuple(i)] == "X":
            count1 += 1
    for i in player1ships[4][1:]:
        if M1[tuple(i)] == "X":
            count2 += 1
    if count1 == 4 and count2 != 4 or count1 != 4 and count2 == 4:
        print("Battleship  X -".ljust(28), end="")
        f.write("Battleship  X -".ljust(28))
    if count1 != 4 and count2 != 4:
        print("Battleship  - -".ljust(28), end="")
        f.write("Battleship  - -".ljust(28))
    if count1 == 4 and count2 == 4:
        print("Battleship  X X".ljust(28), end="")
        f.write("Battleship  X X".ljust(28))

    count1, count2 = 0, 0
    for i in player2ships[3][1:]:
        if M2[tuple(i)] == "X":
            count1 += 1
    for i in player2ships[4][1:]:
        if M2[tuple(i)] == "X":
            count2 += 1
    if count1 == 4 and count2 != 4 or count1 != 4 and count2 == 4:
        print("Battleship  X -")
        f.write("Battleship    X -\n")
    if count1 != 4 and count2 != 4:
        print("Battleship  - -")
        f.write("Battleship    - -\n")
    if count1 == 4 and count2 == 4:
        print("Battleship  X X")
        f.write("Battleship    X X\n")

    count = 0
    for i in player1ships[0][1:]:
        if M1[tuple(i)] == "X":
            count += 1
    if count == 3:
        print("Destroyer   X".ljust(28), end="")
        f.write("Destroyer   X".ljust(28))
    else:
        print("Destroyer   -".ljust(28), end="")
        f.write("Destroyer   -".ljust(28))
    count = 0
    for i in player2ships[0][1:]:
        if M2[tuple(i)] == "X":
            count += 1
    if count == 3:
        print("Destroyer   X")
        f.write("Destroyer X\n")
    else:
        print("Destroyer   -")
        f.write("Destroyer -\n")

    count = 0
    for i in player1ships[1][1:]:
        if M1[tuple(i)] == "X":
```

```
            count += 1
    if count == 3:
        print("Submarine   X".ljust(28), end="")
        f.write("Submarine   X".ljust(28))
    else:
        print("Submarine   -".ljust(28), end="")
        f.write("Submarine   -".ljust(28))
    count = 0
    for i in player2ships[1][1:]:
        if M2[tuple(i)] == "X":
            count += 1
    if count == 3:
        print("Submarine   X")
        f.write("Submarine X\n")
    else:
        print("Submarine   -")
        f.write("Submarine -\n")
    count1, count2, count3, count4 = 0, 0, 0, 0
    for i in player1ships[5][1:]:
        if M1[tuple(i)] == "X":
            count1 += 1
    for i in player1ships[6][1:]:
        if M1[tuple(i)] == "X":
            count2 += 1
    for i in player1ships[7][1:]:
        if M1[tuple(i)] == "X":
            count3 += 1
    for i in player1ships[8][1:]:
        if M1[tuple(i)] == "X":
            count4 += 1
    counts = [count1, count2, count3, count4]
    swims = 0
    sunk = 0
    for count in counts:
        if count == 2:
            sunk += 1
        else:
            swims += 1
    if sunk == 4:
        print("Patrol Boat X X X X".ljust(28), end="")
        f.write("Patrol Boat X X X X".ljust(28))
    if sunk == 3:
        print("Patrol Boat X X X -".ljust(28), end="")
        f.write("Patrol Boat X X X -".ljust(28))
    if sunk == 2:
        print("Patrol Boat X X - -".ljust(28), end="")
        f.write("Patrol Boat X X - -".ljust(28))
    if sunk == 1:
        print("Patrol Boat X - - -".ljust(28), end="")
        f.write("Patrol Boat X - - -".ljust(28))
    if sunk == 0:
        print("Patrol Boat - - - -".ljust(28), end="")
        f.write("Patrol Boat - - - -".ljust(28))
    count1, count2, count3, count4 = 0, 0, 0, 0
    for i in player2ships[5][1:]:
        if M2[tuple(i)] == "X":
            count1 += 1
    for i in player2ships[6][1:]:
        if M2[tuple(i)] == "X":
            count2 += 1
    for i in player2ships[7][1:]:
        if M2[tuple(i)] == "X":
            count3 += 1
```

```
    for i in player2ships[8][1:]:
        if M2[tuple(i)] == "X":
            count4 += 1

    counts = [count1, count2, count3, count4]
    swims = 0
    sunk = 0
    for count in counts:
        if count == 2:
            sunk += 1
        else:
            swims += 1
    if sunk == 4:
        print("Patrol Boat X X X X")
        f.write("Patrol Boat   X X X X\n")
    if sunk == 3:
        print("Patrol Boat X X X -")
        f.write("Patrol Boat   X X X -\n")
    if sunk == 2:
        print("Patrol Boat X X - -")
        f.write("Patrol Boat   X X - -\n")
    if sunk == 1:
        print("Patrol Boat X - - -")
        f.write("Patrol Boat   X - - -\n")
    if sunk == 0:
        print("Patrol Boat - - - -")
        f.write("Patrol Boat   - - - -\n")
    f.write("\n")
```

This function prints the ship counter above the boards. It checks the ship dictionaries and counts the squares of each ship that have been hit. If a ship is fully hit, it prints an **X**; otherwise, it prints a **-** for the ship. String concatenation is not used; instead, the **end=""** method is used to keep printing to the same line.

After this function, it is time to play rounds. A try-except block checks the player's inputs and raises an index error according to the number of **;** in the line. If the letter between **;** is not a predetermined character, it raises a value error.

Function_matrixer(dictname)

Alph list is created to create a dictionary matrix for the board every square is set to – at first

Matrix 2 and 1 are created

```
def matrixer(dictname):
    for num in range(1, 11):
        for let in alph:
            dictname[str(num), let] = "-"


matrixer(M2)
matrixer(M1)
```

Function_Moves(in1,in2,ct)

```
def moves(in1, in2, ct1=0):
    print("Battle of Ships Game\n")  # starting sentence
    f.write("Battle of Ships Game\n\n")  # starting sentence
    # ct is used to take a count of the rounds
    # if error is raised in return function called with mines ct1-1
because each call adds 1 to ct1
    # also if it were not called like ct-1 unvalid operands would
ruin rounds
    ct1 += 1

    # if the returned list is empty print player is out of moves
    if not in1:
        print("Player1 is out of moves")
        f.write("Player1 is out of moves")
    elif not in2:
        print("Player2 is out of moves")
        f.write("Player2 is out of moves")
    # first check if the operand is valid!
    else:
        # 2,A operand to ["2","A"]
        P1moves = in1[0].split(",")
        P2moves = in2[0].split(",")
        try:
            # checking for empty and missing item operands
            if "" in P1moves or len(P1moves) == 1:
                raise IndexError(P1moves)
            if "" in P2moves or len(P2moves) == 1:
                raise IndexError(P2moves)
            # checking if first part of move operand is integer or
not!
            try:
                P1moves[0] != int(P1moves[0])
            except ValueError:
                print("ValueError: operand of Player1 {} is invalid,
Next operand will be used! \n".format(P1moves))
                f.write("ValueError: operand of Player1 {} is
invalid, Next operand will be used! \n".format(P1moves))
                return moves(in1[1:], in2[:], ct1 - 1)
            try:
                P2moves[0] != int(P2moves[0])
            except ValueError:
                print("ValueError: operand of Player2 {} is invalid,
Next operand will be used!\n".format(P2moves))
```

```
                f.write("ValueError: operand of Player2 {} is
invalid, Next operand will be used!\n".format(P2moves))
                return moves(in1[:], in2[1:], ct1 - 1)
            # checking if operand is longer then expected
            if len(P1moves) != 2:
                raise ValueError(P1moves)
            if len(P2moves) != 2:
                raise ValueError(P2moves)
            # checking to see if the second part of the move is a
letter or not
            if P1moves[1] not in alphabet:
                raise ValueError("C")
            if P2moves[1] not in alphabet:
                raise ValueError("D")
            # checking to see if the second part of the move is a
letter which is meaningful(A,B,C,D...I,J) or not
            if P1moves[1] not in alph:
                raise AssertionError("1")
            if P2moves[1] not in alph:
                raise AssertionError("2")
            # checking to see if the first part of the move is a
number which is meaningful(1,2,3,...9,10) or not

            if int(P2moves[0]) > 10 or int(P2moves[0]) < 0:
                raise AssertionError("3")
            if int(P1moves[0]) < 0 or int(P1moves[0]) < 0:
                raise AssertionError("4")

        except IndexError as e:
            if e.args[0] == P1moves:
                print("IndexError operand of Player1 {} is missing
arguments, Next operand will be used!\n".format(
                    P1moves))
                f.write("IndexError operand of Player1 {} is missing
arguments, Next operand will be used!\n".format(
                    P1moves))
                return moves(in1[1:], in2[:], ct1 - 1)
            if e.args[0] == P2moves:
                print("IndexError operand of Player2 {} is missing
arguments, Next operand will be used!\n".format(
                    P2moves))
                f.write("IndexError operand of Player2 {} is missing
arguments, Next operand will be used!\n".format(
                    P2moves))

                return moves(in1[:], in2[1:], ct1 - 1)
        except ValueError as e:
            if e.args[0] == P1moves:
                print("ValueError: operand of Player1 {} is invalid,
Next operand will be used!\n".format(P1moves))
                f.write("ValueError: operand of Player1 {} is
invalid, Next operand will be used!\n".format(P1moves))
                return moves(in1[1:], in2[:], ct1 - 1)
            if e.args[0] == P2moves:
                print("ValueError: operand of Player2 {} is invalid,
Next operand will be used!\n".format(P2moves))
                f.write("ValueError: operand of Player2 {} is
invalid, Next operand will be used!\n".format(P2moves))
                return moves(in1[:], in2[1:], ct1 - 1)
            if e.args[0] == "D":
                print("ValueError: operand of Player2 {} is invalid,
Next operand will be used!\n".format(P2moves))
                f.write("ValueError: operand of Player2 {} is
```

```
invalid, Next operand will be used!\n".format(P2moves))
                return moves(in1[:], in2[1:], ct1 - 1)
            if e.args[0] == "C":
                print("ValueError: operand of Player1 {} is invalid,
Next operand will be used!\n".format(P1moves))
                f.write("ValueError: operand of Player1 {} is
invalid, Next operand will be used!\n".format(P1moves))
                return moves(in1[1:], in2[:], ct1 - 1)
        except AssertionError as j:
            if j.args[0] == "1" or j.args[0] == "4":
                f.write("AssertionError: Invalid Operation\n")
                print("AssertionError: Invalid Operation")
                return moves(in1[1:], in2[:], ct1 - 1)
            if j.args[0] == "2" or j.args[0] == "3":
                print("AssertionError: Invalid Operation")
                f.write("AssertionError: Invalid Operation\n")
                return moves(in1[:], in2[1:], ct1 - 1)
        except  Exception:
            print("kaBOOM: run for your life!")
            f.write("kaBOOM: run for your life!")
            quit()
        print("Player1's Move\n")
        f.write("Player1's Move\n\n")
        starter(ct1)
        lister(M1, M2)
        ships()
        print(" ")
        print("Enter your move: {}\n".format(",".join(P1moves)))
        f.write("Enter your move: {}\n\n".format(",".join(P1moves)))

        # register the hit | player1 hit player2
        for ship in player2ships:  # p1 hits p2 creates p2's table
            if P1moves in ship:
                dictvar = tuple(P1moves)
                M2[dictvar] = "X"
                break
            else:
                dictvar = tuple(P1moves)
                M2[dictvar] = "O"
        # print the previous hit results in table
        print("Player2's Move\n")
        f.write("Player2's Move\n\n")
        starter(ct1)
        lister(M1, M2)
        ships()

        print("\nEnter your move: {}\n".format(",".join(P2moves)))
        f.write("Enter your move: {}\n\n".format(",".join(P2moves)))

        # register the hit | player2 hit player1

        for ship in player1ships:
            dictvar = tuple(P2moves)

            # p2 hits p1 creates p1's table
            if P2moves in ship:
                M1[dictvar] = "X"
                break
            else:
                M1[dictvar] = "O"
        # round is done p2's effect will be shown in next round

        # checking the amount of each player's ship floating left to
```

```
call winner function
        # if one of players ships all sunk declare who have won
        # if each has zero floating ship square left declare draw!
        count1 = 0

        for ship in player2ships:
            for sqr in ship[1:]:
                if M2[tuple(sqr)] != "X":
                    count1 += 1

        count2 = 0
        for ship in player1ships:
            for sqr in ship[1:]:
                if M1[tuple(sqr)] != "X":
                    count2 += 1

        if count1 == 0 and count2 == 0:
            tmp4 = "It is a draw!\n\n" + "Final Information\n\n" +
"Player1's Board\t\t\t\tPlayer2's Board\n" + "  A B C D E F G H I
J\t\t  A B C D E F G H I J\n"
            tmp3 = "It is a draw!\n\n" + "Final Information\n\n" +
"Player1's Board\t\t\t\tPlayer2's Board\n" + "  A B C D E F G H I
J\t\t  A B C D E F G H I J"
            print(tmp3)
            f.write(tmp4)
            winner(M1, M2)
        elif count1 == 0:
            a = "Player1 Wins!\n\n" + "Final Information\n\n" +
"Player1's Board\t\t\t\tPlayer2's Board\n" + "  A B C D E F G H I
J\t\t  A B C D E F G H I J\n"
            b = "Player1 Wins!\n\n" + "Final Information\n\n" +
"Player1's Board\t\t\t\tPlayer2's Board\n" + "  A B C D E F G H I
J\t\t  A B C D E F G H I J"
            print(b)
            f.write(a)
            winner(M1, M2)
        elif count2 == 0:
            a = "Player2 Wins!\n\n" + "Final Information\n\n" +
"Player1's Board\t\t\t\tPlayer2's Board\n" + "  A B C D E F G H I
J\t\t  A B C D E F G H I J\n"
            b = "Player2 Wins!\n\n" + "Final Information\n\n" +
"Player1's Board\t\t\t\tPlayer2's Board\n" + "  A B C D E F G H I
J\t\t  A B C D E F G H I J"
            print(b)
            f.write(a)
            winner(M1, M2)

        return moves(in1[1:], in2[1:], ct1)
```

This function processes the rounds in a recursive manner. Nearly all the other functions are used or built for this main function.

An alphabet list is created to check for errors. The function uses an optional argument for round counting. It prints the starting line of the game.

The base case checks if the returned lists have any items to continue. If they are not empty, the first element of each list is split and assigned as **P1_moves** and **P2_moves**.

A try-except block starts to check for faulty inputs. Every faulty operand or move will call the function without itself, but the first element of the other list will remain, allowing the recursion to continue without losing an unfaulty operand.

The function first checks if the operand is empty and the length of the operand. If either check fails, it raises an index error. It then tries to convert the first element of the operand to an integer to check if it is a numeric string, as it should be. The function then checks the length of the operands again.

Next, it checks if the second part is in the alphabet list. If it is not, it raises a value error. It then checks if the length of the operand is 3. If it is, it checks if the third part is in the alphabet list. If it is not, it raises a value error.

The function then calls **converter()** on the second and third parts of the operand to get the numeric values. It checks if the numeric values are within the range of the board. If either value is not within the range, it raises an index error.

The function then checks if the board square indicated by the operand is already hit. If it is, it raises a value error. If the square is not hit, the function marks it as hit in the appropriate player's board dictionary.

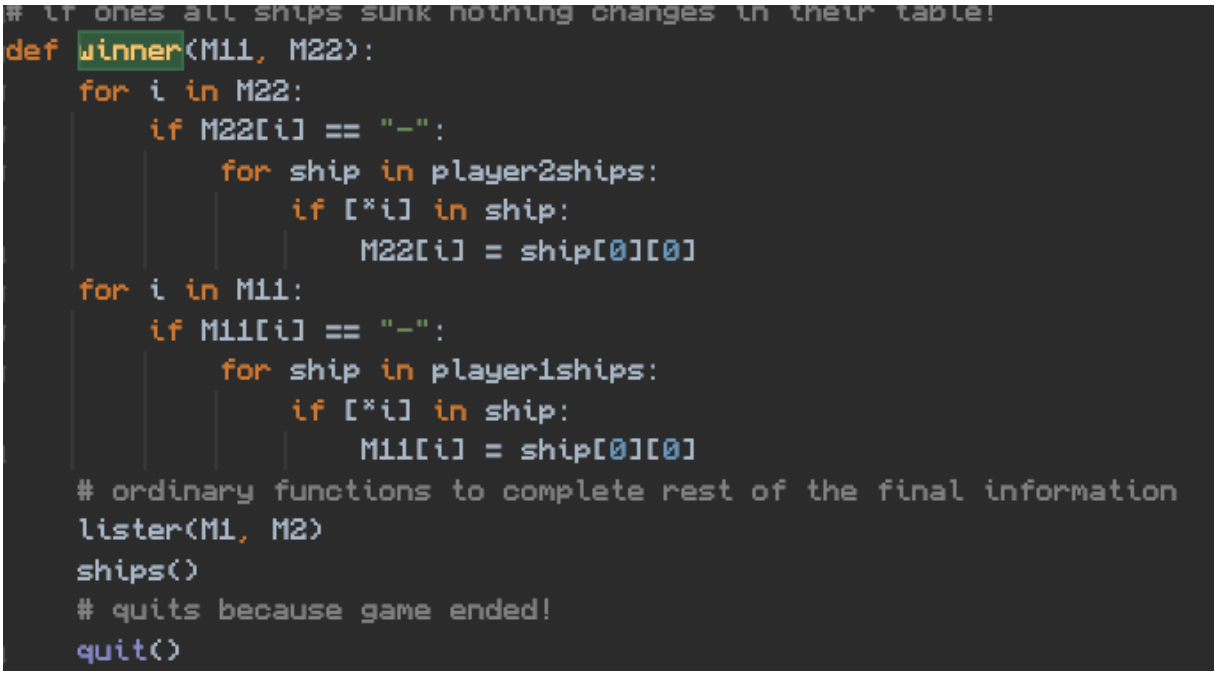
The function then checks if the move sinks a ship. If it does, it prints a message and removes the sunk ship from the player's ship dictionary.

After all the error checking and updating of the board and ship dictionaries, the function prints the updated board and ship counters.

The function then calls itself with the updated lists of moves for the next round.

Function_Winner

The function first iterates through the elements of **M22** and checks if any of them have a value of "-". If it finds such an element, it looks for a ship belonging to the second player that includes that element's coordinates in its list of positions. If it finds a match, it replaces the "-" with the first letter of the ship's name. The function then does the same thing for **M11** and the ships belonging to the first player. After this, the function appears to call two other functions, **lister** and **ships.**



```
# if ones all ships sunk nothing changes in their table!
def winner(M11, M22):
    for i in M22:
        if M22[i] == "-":
            for ship in player2ships:
                if [*i] in ship:
                    M22[i] = ship[0][0]
    for i in M11:
        if M11[i] == "-":
            for ship in player1ships:
                if [*i] in ship:
                    M11[i] = ship[0][0]
    # ordinary functions to complete rest of the final information
    lister(M1, M2)
    ships()
    # quits because game ended!
    quit()
```

Programmer's Catalogue

I spent approximately 18 hours on this project. I spent the first 12 hours creating the base of the code, and the remaining 6 hours fixing cosmetic issues. I believe my code is useful because the main components are separated into different functions and the code is well-organized.

User's Catalogue

To use this code, create text files that follow the rules outlined in the provided PDF. Then, run the code by entering "python3 Assignment4.py Player1.txt Player2.txt Player1.in Player2.in" into the terminal. You can view the results in the terminal and in the "battleship.out" file.