Hacettepe University
BBM103 Assignment:4 Battle of Ships

Süleyman Yolcu – 2210765016

25.12.2022

Index

1-Analysis

Battleship game is a two-player strategic board game played on a grid. Each player has a set of ships of different sizes that they place on their own grid, and the goal of the game is to sink all of the opponent's ships before they sink all of yours. The players take turns making attacks on the other player's grid by specifying a coordinate on the grid, and the opponent responds by indicating whether or not the attack was a hit or a miss. The game ends when all of the ships on one player's board have been sunk.

2-Design

This code is a Python script that plays the game "Battleship". It reads in four command line arguments: two text files representing the boards of two players, and two input files representing the ships and attacks of the two players. The script then simulates a game of Battleship between the two players by reading in the board layouts, placing the ships on the board, and simulating attacks between the players. The game continues until one player has no more ships left, at which point the script writes the result of the game to an output file called "Battleship.out".

The script defines several dictionaries to store information about the game. The **player_open_boards** dictionary stores the board layout of each player, with their ships visible. The **player_hidden_boards** dictionary stores the board layout of each player, with their ships hidden. The **player_ships** dictionary stores the locations of each ship on the board for each player. The **player_ship_status** dictionary stores the status (sunk or not sunk) of each ship for each player. The **missing_files** list stores the IOError causing files. The **message** string stores the error messages.

The script defines custom exceptions called **ShipOverlapError, DuplicatemoveError** which is raised if two ships overlap on the board and raised if a move played twice. It also defines a function **save_output** which writes a message to the output file and prints it to the console.

The script defines a function **create_board** which reads in a text file representing the board layout for a player and creates a dictionary representing the board. The function first initializes an empty dictionary called **board_dict** which will be used to store the board layout. It then loops through each row and column of the board, and initializes each cell to contain a dash ("-"). The function then opens the input text file and reads in the ship positions. It loops through each position and checks if it is a ship or an empty space. If it is a ship, it adds the ship to the board layout in **board_dict**. If it is an empty space, it does nothing.

The function then stores the board layout in the **player_open_boards** dictionary using the player's name as the key. The function also creates the **player_hidden_boards** dictionary for to be used in **battle_of_ships** funciton.

The script defines a function **ship_locations** which reads in input files representing the ship locations and creates a dictionary storing the locations of each ship on the board. The function first initializes a dictionary called **ships** which will be used to store the ship locations. It then loops through the **player_open_boards** dictionary and adds the locations of the ships "C", "D", and "S" to the **ships** dictionary. It then opens the input file and reads in the ship names and positions. It processes the data and stores the locations of each ship in the **ships** dictionary.

The script defines a function **show_hidden_board** which shows the players hidden boards during the game.

The script defines a funciton **ship_status_show** which takes the data from **player_hidden-boards** dictionary and shows the current status of the player ships sunk or floating.

Before getting into the **battle_of_ships** funciton I want to mention two helpful funciton. First one is **convert** funciton which takes the move string and converts to a tuple for to be used as a dictionary key. Second one is **input_checker** funciton which checks if the given attack move input is valid or not.

The script defines a function **battle_of_ships** which simulates an attack by a player on another player's board. The function takes in the name of the attacking player and the location of the attack, and checks if the attack was a hit or a miss. If it was a hit, the function updates the board layout in the **player_hidden_boards** dictionary and the ship status in the **player_ship_status** dictionary. If it was a miss, the function does nothing.

The attack loop continues until one player has no more ships left. At the beginning of each iteration of the loop, the script reads in the next attack from each player's input file and simulate the attack. If a player's ship is sunk, the script updates the **player_ship_status** dictionary to reflect that the ship is sunk.

After all attacks have been processed, the script checks if one player has no more ships left. If this is the case, the script writes the result of the game to the output file and breaks out of the loop. If both players still have ships, the loop continues and the next set of attacks is processed.

Finally, the script writes the winner and closes the output file and exits.

The script also defines a **main** funciton which makes multiple error handling and calls the neccesary functions by order and simulates the whole game.

3-Programmer’s Catalouge

Time spent analyzing

Unfortunately, I did not know Battleship game but I learned quickly cause it is a simple game. Although the game is simple, coding was not easy as the game. As I mentioned, I learned the game and the game rules quickly so I can say I spent 15 minutes to fully understand the game and the rules.

Time spent designing

While designing the code, I wrote down the steps that I should follow. First, I needed to create two boards for each player. One to place ships, one to use while battle. Second, I needed to specify the locations of the ships in order to simplicity of the script. Last but not least, I needed to read the moves and start the game. At the beginning it looks simple. Yet, I learned in the future that I will have to design a lot of other things. Overall, I spent about 1 and a half hours designing the script.

Time spent implementing

This part took the longest. While implementing I have encountered many issues. For example: Locating the ships, holding data whereas its reachable, error handling… After implementing the main parts (creating boards and locating the ships and their names), I tried to make a sufficent attacking funciton because this functions was where the real magic happens. After defining three funciton (convert, show_hidden_board, input_checker) most of the implementing completed. Overall, I spent about 7-8 hours in this part.

Time spent testing, reporting

While testing, I tested nearly every possible scenario which includes: enough system arguments, correctness of input files, correctnes of valid move operations. This part took about 2 hours in total. In reporting part, I wanted to explicitly explain my code and reporting part took about 2 hours too.

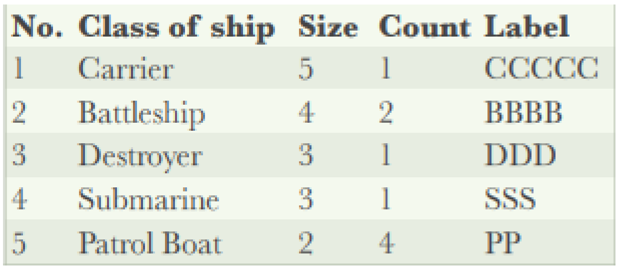
Code

```python
# Süleyman Yolcu b2210765016
import sys

output_file = open("Battleship.out", "w")
player_open_boards = {}
player_hidden_boards = {}
player_ships = {}
player_ship_status = {}
missing_files = []
message = ""


class ShipOverlapError(Exception):
    pass


class DuplicateMoveError(AssertionError):
    pass


# This function creates both open boards for players
def create_board(input_file):
    player = input_file.strip(".txt")
    board_dict = {}
    for row in range(1, 11):
        for column in range(65, 75):
            board_dict[(str(row), chr(column))] = "-"
    player_hidden_boards[player] = dict(board_dict)
    with open(input_file, "r") as input_file:  # Locating the ships
        ship_positions = input_file.read().splitlines()
        for index1, position in enumerate(ship_positions):
            for index2, element in enumerate(position):
                if element != ";":
                    if index2 + 1 != len(position) and position[index2 + 1]
!= ";":  # Checking if ship overlaps
                        raise ShipOverlapError
                    elif index2 + 1 == len(position) and position[index2 -
1] != ";":  # Checking if ship overlaps
                        raise ShipOverlapError
# for last columns
                    counter = position[:index2].count(";")
                    board_dict[(str(index1 + 1), chr(65 + counter))] =
element
        player_open_boards[player] = board_dict


# This funciton holds the name and position of every ship for players.
def ship_locations(input_file):
    ships = {"C1": [], "B1": [], "B2": [], "D1": [], "S1": [], "P1": [],
"P2": [], "P3": [], "P4": []}
    player = input_file.rstrip(".txt").lstrip("Optional")
    for i in player_open_boards[player].items():  # Taking positions of
single ships
        if i[1] == "C":
            ships["C1"].append(i[0])
        elif i[1] == "D":
            ships["D1"].append(i[0])
        elif i[1] == "S":
            ships["S1"].append(i[0])
    with open(input_file, "r") as f:  # Taking positions of multiple ships
        for i in f:
            ship_name = i[:2]
            position = i[3:]
            point_direction = position.rstrip(";\n").split(";")
            data = [ship_name] + point_direction
            start = data[1].split(",")
            if data[2] == "right":
                if data[0].startswith("B"):
                    for j in range(ord(start[1]), ord(start[1]) + 4):
                        ships[data[0]].append((start[0], chr(j)))
                if data[0].startswith("P"):
                    for j in range(ord(start[1]), ord(start[1]) + 2):
                        ships[data[0]].append((start[0], chr(j)))
            if data[2] == "down":
                if data[0].startswith("B"):
                    for j in range(int(start[0]), int(start[0]) + 4):
                        ships[data[0]].append((str(j), start[1]))
                if data[0].startswith("P"):
                    for j in range(int(start[0]), int(start[0]) + 2):
                        ships[data[0]].append((str(j), start[1]))
        player_ships[player] = ships


# This funciton shows the hidden boards in every round.
def show_hidden_board():
    lst1 = list(player_hidden_boards["Player1"].values())  # Making a 1D
list
    lst2 = list(player_hidden_boards["Player2"].values())
    save_output(f"  A B C D E F G H I J         A B C D E F G H I J")
    for i in range(1, 11):  # Displaying 1D list like boards
        print(f"{str(i)} " if i < 10 else f"{str(i)}", end="")
        output_file.write(f"{str(i)} " if i < 10 else f"{str(i)}")
        for j in range(10):
            index = ((i - 1) * 10) + j
            print(f"{lst1[index]} " if j != 9 else f"{lst1[index]}",
end="")
            output_file.write(f"{lst1[index]} " if j != 9 else
f"{lst1[index]}")
        print(f"\t\t", end="")
        output_file.write(f"\t\t")
        print(f"{str(i)} " if i < 10 else f"{str(i)}", end="")
        output_file.write(f"{str(i)} " if i < 10 else f"{str(i)}")
        for k in range(10):
            index = ((i - 1) * 10) + k
            print(f"{lst2[index]} " if k != 9 else f"{lst2[index]}",
end="")
            output_file.write(f"{lst2[index]} " if k != 9 else
f"{lst2[index]}")
        save_output("")
    save_output("")
    ship_status_show()  # Displaying number of ships sunk / floating


# This funciton shows the number of sunk and floating ships in every round.
def ship_status_show():
    for y in ["Player1", "Player2"]:
        ship_status = {"Carrier": ["-"], "Battleship": ["-", "-"],
"Destroyer": ["-"], "Submarine": ["-"],
                       "Patrol Boat": ["-", "-", "-", "-"]}
        c1 = [player_hidden_boards[y].get(i) for i in
player_ships[y]["C1"]]  # Collecting statuses of ships
        b1 = [player_hidden_boards[y].get(i) for i in
player_ships[y]["B1"]]  # sunk or floating
        b2 = [player_hidden_boards[y].get(i) for i in
player_ships[y]["B2"]]
        d1 = [player_hidden_boards[y].get(i) for i in
player_ships[y]["D1"]]
        s1 = [player_hidden_boards[y].get(i) for i in
player_ships[y]["S1"]]
        p1 = [player_hidden_boards[y].get(i) for i in
player_ships[y]["P1"]]
        p2 = [player_hidden_boards[y].get(i) for i in
player_ships[y]["P2"]]
        p3 = [player_hidden_boards[y].get(i) for i in
player_ships[y]["P3"]]
        p4 = [player_hidden_boards[y].get(i) for i in
player_ships[y]["P4"]]
        if all(x == "X" for x in c1):  # Controlling if all the parts of
the ships
            ship_status["Carrier"] = ["X"]  # are sunk
        if all(x == "X" for x in b1):
            ship_status["Battleship"].insert(0, "X")
            ship_status["Battleship"].pop()
        if all(x == "X" for x in b2):
            ship_status["Battleship"].insert(0, "X")
            ship_status["Battleship"].pop()
        if all(x == "X" for x in d1):
            ship_status["Destroyer"] = ["X"]
        if all(x == "X" for x in s1):
            ship_status["Submarine"] = ["X"]
        if all(x == "X" for x in p1):
            ship_status["Patrol Boat"].insert(0, "X")
            ship_status["Patrol Boat"].pop()
        if all(x == "X" for x in p2):
            ship_status["Patrol Boat"].insert(0, "X")
            ship_status["Patrol Boat"].pop()
        if all(x == "X" for x in p3):
            ship_status["Patrol Boat"].insert(0, "X")
            ship_status["Patrol Boat"].pop()
        if all(x == "X" for x in p4):
            ship_status["Patrol Boat"].insert(0, "X")
            ship_status["Patrol Boat"].pop()
        player_ship_status[y] = ship_status
    for P1, P2 in zip(player_ship_status["Player1"],
player_ship_status["Player2"]):  # Displaying the table
        x = int((12 - len(P1)) / 4 + 1)
        p1 = str(' '.join(map(str, player_ship_status['Player1'][P1])))
        y = int((16 - len(p1)) / 4 + 1)
        save_output(
            f"{P1:\t<{len(P1) + x}}{p1:\t<{len(p1) + y}}{P2:\t<{str(len(P1)
+ x)}}{' '.join(map(str, player_ship_status['Player2'][P2]))}")
    save_output("")


# This funciton converts moves to tuples in order to use for dictionary
keys.
def convert(string):
    t = tuple(string.split(","))
    return t
```

```
# This is the magical funciton which starts the game.
def battle_of_ships():
    board_dict = {}  # Creating hidden boards
    for row in range(1, 11):
        for column in range(65, 75):
            board_dict[(str(row), chr(column))] = "-"
    player_hidden_boards["Player1"] = dict(board_dict)
    player_hidden_boards["Player2"] = dict(board_dict)
    save_output(f"Battle of Ships Game")
    with open("Player1.in", "r") as f1, open("Player2.in", "r") as f2:
        contents1 = f1.read()
        contents2 = f2.read()
        contents1 = contents1.replace("\n", "").rstrip(";").split(";")
        contents2 = contents2.replace("\n", "").rstrip(";").split(";")
        i = 0
        j = 0
        r = 1
        result = ""
        while True:
            save_output(f"\nPlayer1's Move\n")  # Player1's Turn
            save_output(f"Round : {r}\t\t\t\t\tGrid Size: 10x10\n")
            save_output(f"Player1's Hidden Board\t\tPlayer2's Hidden
Board")
            show_hidden_board()
            while i < len(contents1):
                save_output(f"Enter your move: {contents1[i]}")
                try:  # Checking if the move is valid or not.
                    input_checker(contents1[i])
                    if
player_hidden_boards["Player2"][convert(contents1[i])] != "-":
                        raise DuplicateMoveError("AssertionError: Invalid
Operation.")
                except IndexError:
                    i += 1
                    save_output(f"Index Error: {message}")
                except ValueError:
                    i += 1
                    save_output(f"Value Error: {message}")
                except DuplicateMoveError as e:
                    i += 1
                    save_output(f"{e}")
                except AssertionError:
                    i += 1
                    save_output(f"Assertion Error: {message}")
                else:
                    if player_open_boards["Player2"][convert(contents1[i])]
== "-":  # Attack part

player_hidden_boards["Player2"][convert(contents1[i])] = "O"
                    elif
player_open_boards["Player2"][convert(contents1[i])] != "-":

player_hidden_boards["Player2"][convert(contents1[i])] = "X"
                    i += 1
                    break
            save_output("\nPlayer2's Move\n")  # Player2's Turn
            save_output(f"Round : {r}\t\t\t\t\tGrid Size: 10x10\n")
            save_output(f"Player1's Hidden Board\t\tPlayer2's Hidden
Board")
            show_hidden_board()
            while j < len(contents2):
                save_output(f"Enter your move: {contents2[j]}")
                try:  # Checking if the move is valid or not.
                    input_checker(contents2[j])
                    if
player_hidden_boards["Player1"][convert(contents2[j])] != "-":
                        raise DuplicateMoveError("AssertionError: Invalid
Operation.")
                except IndexError:
                    j += 1
                    save_output(f"Index Error: {message}")
                except ValueError:
                    j += 1
                    save_output(f"Value Error: {message}")
                except DuplicateMoveError as e:
                    j += 1
                    save_output(f"{e}")
                except AssertionError:
                    j += 1
                    save_output(f"Assertion Error: {message}")
                else:
                    if player_open_boards["Player1"][convert(contents2[j])]
== "-":  # Attack part

player_hidden_boards["Player1"][convert(contents2[j])] = "O"
                    elif
player_open_boards["Player1"][convert(contents2[j])] != "-":

player_hidden_boards["Player1"][convert(contents2[j])] = "X"
                    j += 1
                    break
            p1 = list(player_hidden_boards["Player1"].values())
            p2 = list(player_hidden_boards["Player2"].values())
            if p1.count("X") == 27 and p2.count("X") != 27:  # Player2 wins
                result = result + "Player2 Wins!"
                winner = ["Player2"]
                break
            elif p1.count("X") != 27 and p2.count("X") == 27:  # Player1
wins
                result = result + "Player1 Wins!"
                winner = ["Player1"]
                break
            elif p1.count("X") == 27 and p2.count("X") == 27:  # Draw
                result = result + "It is a Draw!"
                winner = ["Player1", "Player2"]
                break
            elif (i == len(contents1) or j == len(contents2)) and
(p1.count("X") != 27 and p2.count("X") != 27):
                result = result + "The game did not finish with given
inputs!"
                winner = ["Player1", "Player2"]
                break
            r += 1
        save_output(f"\n{result}\n")
        save_output(f"Final Information\n")  # Shows still hidden cells of
unsunk ships of the winner
        for w in winner:
            keys = [key for key, value in player_hidden_boards[w].items()
if value == "-"]
            for key in keys:
                if player_open_boards[w][key] != "-":
                    letter = player_open_boards[w][key]
                    player_hidden_boards[w][key] = letter
            keys.clear()
        save_output(f"Player1's Board\t\t\t\tPlayer2's Board")
        show_hidden_board()


# This funciton checks the input, if it is not valid it throws specific
exception
def input_checker(value):
    global message
    lst = value.split(",")
    if len(value) < 3:
        if len(value) == 2:
            if lst[0] == "":
                message = "Missing x coordinate!"
                raise IndexError
            if lst[1] == "":
                message = "Missing y coordinate!"
                raise IndexError
        if value.isalpha():
            message = "You must enter X coordinate and comma!"
            raise IndexError
        if value.isdigit():
            message = "You must enter comma and Y coordinate!"
            raise IndexError
        if value == "":
            message = "You must enter coordinates separated by comma!"
            raise IndexError
        if value == ",":
            message = "You must enter X and Y coordinates!"
            raise IndexError
    elif lst[0].isalpha() and lst[1].isdigit():
        message = "X coordinate must be integer! Y coordinate must be
letter!"
        raise ValueError
    elif lst[0].isalpha():
        message = "X coordinate must be integer!"
        raise ValueError
    elif lst[1].isdigit():
        message = "Y coordinate must be integer!"
        raise ValueError
    elif len(lst) > 2:
        message = "Too many arguments!"
        raise ValueError
    elif int(lst[0]) > 10 and ord(lst[1]) > 74:
        message = "Invalid Operation."
        raise AssertionError
    elif int(lst[0]) > 10:
        message = "Invalid Operation."
        raise AssertionError
    elif ord(lst[1]) > 74:
        message = "Invalid Operation."
        raise AssertionError


# This function displays the output
def save_output(x):
    output_file.write(f"{x}\n")
    print(x)
```

```python
# This funciton runs the code
def main():
    try:  # Checked if there are enough arguments
        assert len(sys.argv) == 5
    except AssertionError:
        save_output("Not enough arguments !")
        sys.exit()
    try:  # Checking every file
        create_board(sys.argv[1])
    except IOError:
        missing_files.append(sys.argv[1])
    except ShipOverlapError:
        save_output("kaBOOM: run for your life!")
        sys.exit()
    try:
        create_board(sys.argv[2])
    except IOError:
        missing_files.append(sys.argv[2])
    except ShipOverlapError:
        save_output("kaBOOM: run for your life!")
        sys.exit()
    try:
        with open(sys.argv[3]):
            pass
    except IOError:
        missing_files.append(sys.argv[3])
    try:
        with open(sys.argv[4]):
            pass
    except IOError:
        missing_files.append(sys.argv[4])
    if missing_files:
        prompt = "IOError: input file(s) {} is/are not
reachable.".format(", ".join(missing_files))
        save_output(prompt)
        with open("Battleship.out", "w") as f:
            f.write(prompt + "\n")
        sys.exit()
    ship_locations("OptionalPlayer1.txt")
    ship_locations("OptionalPlayer2.txt")
    battle_of_ships()


main()
output_file.close()
```

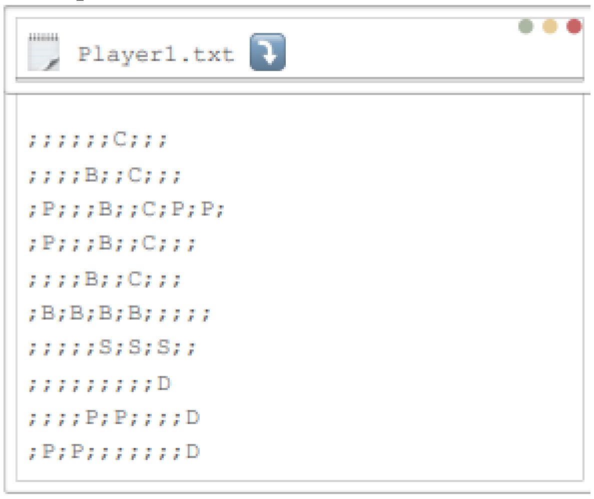
4-User Catalouge



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

- Create two files indicating the ship locations.

  Example file:



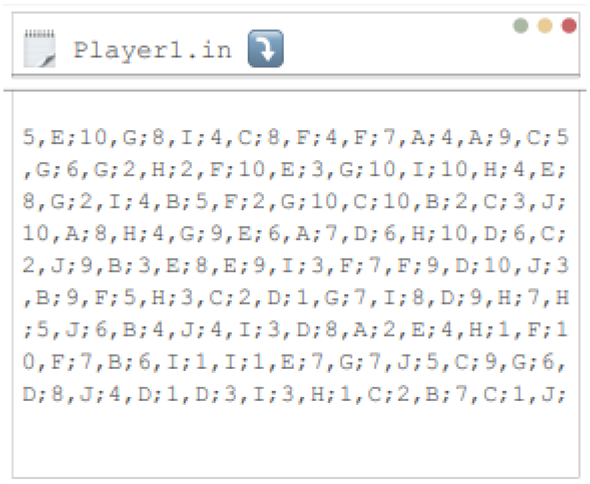
Player1.txt

```
;;;;;;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
```

- Create two files inlcluding the moves.

  Example file:

move format ({X},{Y})



Player1.in

```
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;1
0,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;
```

Evaluation	Points	Evaluate Yourself / Guess Grading
Readable Codes and Meaningful Naming	5	5

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
There are several negative evaluations	…	…