Hacettepe University
Department of Computer Engineering
BBM 103

Assignment 4: Battleship

2210356102- Yunus Emre Uluer

3.01.2023

Analysis

Battleship is a game which is played with 2 players. Each player has board which is 10x10. On the board, there are ships. Round by round players shot a square and if there is a ship on that square, program informs the player. In our assignment, player's boards and moves were given as a txt file and we are supposed to simulate a battleship games.

First of all, boards cannot have wrong placements. This is fatally important because game finishes when all the player's ships have sunken. If there is no ship type such as a submarine, the game cannot finish. Also, file errors and missing arguments are also important for the game.

Secondly, errors in the move files should be handled. All the errors have different error messages and if the player writes a wrong move, function should look for another move. This means there is no punishment for the wrong move or impossible move.

Another important thing is the number of moves. Number of moves may change and when all of the moves are played it should be draw if there is no winner.

A Player cannot see their opponent's board. They can see a square if they shoot it. At the end of the game, boards should be shown to the players.

Players are informed when they play their moves, the information about the hidden boards, and the number of remaining ships. Also when they move illegally, information about the sunken ship and result of the game.

Design

Data:

Multi-dimensional arrays can be used for boards. But it may not be the best idea for keeping the data of the ships because ships contain some characters in the array and they are separated from the other ships. We also need to keep the data of the sunken ship so we can create 2-3 lists or dictionaries for making things easier.

Find Ship Function:

The algorithm of the ships can be handled by searching the patterns. So, in the txt file they are organized in a way that if the board is considered as a multi-dimensional array, we can look for a pattern going forward or downward. After we find a ship, we can remove those characters so it cannot confuse the program with overlapped ships. Finally, information on the ships should be kept in the different lists.

Sorting Moves:

Moves of the players can be written wrongly so the sort moves function should be written as a while loop. Condition of the while loop could be the right move of the player. While searching for the right move, it should print error messages if it finds an illegal move. If the parameter is the list of the moves of a player. It should be deleted from the list.

Show Board:

Show board is a function which simply shows the boards of the player and the situation of the game. Boards should be hidden if it is not the final information.

Check-win: According to one of the data's, it checks that every ship of a player has sunken. If every player's ships have sunken result should be a draw. Otherwise, a player wins.

Move:

Move function should get the sorted moves, show the board, and change the data according to the move.

Pseudo-code :

```
While win condition is false :
        Move(player1)
        Move(player2)
        Condition = Checkwin()
```

Programmer’s Catalogue

Time Spent:

Design: Designing the main part of the assignment was the easy part. After that, I thought about the functions I need. While I was writing those functions, I understood that I had to write more of them. So the design was changed a lot during the process of coding. Due to changes, It became problematic. I should have spent more time on the design part. I spent approximately 5 hours on the design part.

Handling errors/Implementing:

Errors were an important part of the assignment, so I tried to be careful about them. After that implementation part was also simple. Reason for its simplicity was the steps have been taken carefully. It took 8 hours.

Testing:

Testing was the most difficult thing in the process of this assignment. I had a hard time testing it because we needed to check everything by hand. I tried to consider every possible case and it was exhausting. It took about 6 hours.

Reporting:

Reporting part took 5.5 hours. Explaining the parts of the code was crucial.

Source Code:

```
import os
import sys
#Yunus Emre Uluer 2210356102

################################################################################

curdir = os.getcwd()
outputpath = os.path.join(curdir, "Battleship.out")

with open(outputpath,'w') as f :
    f.write("Battle of Ships Game\n\n")

# opens outputpath and writes games name


def write(string) :
    with open(outputpath ,'a') as f :
        f.write(f'{string}\n')
        print(string)
```

Write function

```
################################################################################
```

Findship Function

Findship function takes char, its location and size for finding a pattern. Row means +1 in the two dimensional array and column is other list. Returns True, char , end and start. It will be useful for storing the data of ship and deleting it. The alghorithm is a for loop for the size and if patterns match then FoundShip stays True and program finds the ship in the location end and start.+1 -1 is about the indexes of array which start from 0 but moves start from 1.

```
def findship(char,column, row , size ,list):
    # This function takes one char's properties(on the board) and finds a pattern regarding the
char's ship.
    FoundShip = True
    start = column, row
    for i in range(size-1):
        try:
            if  char == list[column][row+1]:
                row+= 1  #looks for start index and end index
            else :
                FoundShip = False
                break

        except IndexError:
            FoundShip = False
            break
    end = column,row
    if FoundShip :
        return True, char,'row',(end[1]-start[1]+1)  #returns True if Ship has found
    else:
        FoundShip = True
        try:
            for i in range(size-1):
                if  char == list[column+1][row]:
                    column+= 1
                else :
                    FoundShip = False
                    break
        except IndexError :
            FoundShip = False
        end = column,row

        if FoundShip :
            return True, char,'column',(end[0]-start[0]+1)
        else :

            return False , char,'column',(end[0]-start[0]+1)
```

SortingBoard Function :

sortingboard function takes list as a parameter which is the txt file of players.Findship function is used in this function. When this function finds a char. It searches if it is a ship or not. List of ships will be the main data type in the program. Lists and dictionaries will be explained later. (Such as Shipsdict)

```
def sortingboard(list):
    list_of_ship= []
    for i in range(10) :
        for j in range(10):
            if list[i][j] in ['B', 'C', 'D', 'S', 'P'] :
                isShip, char, column_or_row, size  =  findship(list[i][j],i,j ,Shipsdict[list[i][j]],list)
                if isShip :
                    list_of_ship.append([char,(i+1,j+1), column_or_row])
                if column_or_row == 'column':
                    for k in range(size) :
                        list[i+k][j] = ''
                else :
                    for k in range(size) :
                        list[i][j+k] = ''
    return list_of_ship
```

Find_Locations function :

This function takes the returned list of a sortingboard. And in a dictionary it appends the all locations to a list. It takes also player information so in dictionaries, program keeps the data of ships in this dictionaries. Example = {player1: [B,(1,A),(2,A),(3,A),(4,A)] …. }
So char in the list will be important for situation function later.

```
def find_locations(list,player):
    global Moveboard
    locations_list = []
    dictionary_list = []
    for  locations in list  :
        char, start , roworcolumn = locations
        shipsort = []
        for  i in   range(Shipsdict[char]):

            if roworcolumn == 'column':
                Playerboard[player][start[0]+i-1][start[1]-1] = char
                shipsort.append((start[0]+i , start[1]))
            else :
                Playerboard[player][start[0]-1][start[1]+i-1] = char
                shipsort.append((start[0] , start[1]+i))
        dictionary_list.append([char,*shipsort.copy()])
    return dictionary_list
```

Lists And Dictionaries

Board1 and Board2 are player's boards(shown). Playerboard is a dictionary. Its values are player2 for board2 and player1 for board1. It is for matching the board with the player. Boards of player will change according to this dictionary later in the code!

Remaining ships string is a string for showing the situation of the game. Program will replace '-'with the 'x' when a ship has sunk. It will be important for the checkwin function also.

Letterdict and reversed one will be used for changing letters into number and vice versa. Ships letters matches characters with the names of the ships.

Boards is a dictionary which holds the locations of the ships and the char of the ship. Example = {player1: [B,(1,A),(2,A),(3,A),(4,A)],[P,(3,E),(3,D)] …. }

Shipsdict gives the size of the ships.

```
board1 = [10*['-'] for i in range(10)]
board2 = [10*['-'] for i in range(10)]
Playerboard = dict(Player1=board1, Player2= board2)


Remaining_ships_string = {'Player1':{'C':'Carrier\t\t-' ,'B':'Battleship\t- -' ,'D':'Destroyer\t-'
,'S':'Submarine\t-' ,'P':'Patrol Boat\t- - - -'},
'Player2':{'C':'Carrier\t\t-' ,'B':'Battleship\t- -' ,'D':'Destroyer\t-' ,'S':'Submarine\t-' ,'P':'Patrol
Boat\t- - - -'}}
Letterdict = {'A': 1, 'B': 2, 'C': 3, 'D': 4, 'E': 5, 'F': 6, 'G': 7, 'H': 8, 'I': 9, 'J': 10}
ReversedLetterdict = dict(zip(Letterdict.values(),Letterdict.keys()))
ShipsLetters = dict(S="Submarine", C='Carrier', P='Patrol Boat', B="Battleship",D='Destroyer')

Boards = dict()
Shipsdict= dict(B=4,C=5,D=3,S=3,P=2)
```

Sortmove Function:

Takes the moves and pops the move from the list. With some try catch methods it looks for if it is illegal move or not if it is legal returns move and the player.

```
def sortmove(moves,player) :
    condition = True
    while condition:
        try:
            move = moves.pop(0).split(',')
            if move == [""] or len(move) == 1 :
                output = ("Index Error: No argument given")
                raise IndexError
            elif move == ["",""]:
                output = ("Index Error: Missing Arguments(letter and number)")
                raise IndexError
            elif move[0] == "":
                output = ("Index Error: Missing Argument(letter)")
                raise IndexError
            elif move[1] == "":
                output = ("Index Error: Missing Argument(number)")
                raise IndexError
            else:
                if move[1].isdigit():
                    output = ("ValueError: second argument must be letter(A-J)")
                    raise ValueError
                elif not(move[0].isdigit()):
                    output = ("ValueError: first argument must be integer(0-9)")
                    raise ValueError
                else:
                    if int(move[0]) > 10 :
                        raise AssertionError
                    try:
                        return((int(move[0])),Letterdict[move[1]]), player
                    except KeyError:
                        raise AssertionError
        except IndexError:
            x = moves[0]
            write(output)
        except ValueError:
            write(output)
        except AssertionError:
            write("AssertionError: Invalid Operation.")
        else :
            condition = False
```

Board Function:

Board function prints the boards of the players. Isfinal is a boolen variable which is False when it is final information. board is not hidden when isFinal is false.

```
def board(isfinal):
    output=(f'  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(10) :
        if i+1 != 10:
            output += str(i+1) + " "
        else :
            output += str(i+1)
        for l in board1[i] :
            if (l not in ['-', 'X' , 'O']) and isfinal:
                output += '-' + " "
            else :
                output += l + " "
        output = output.rstrip(' ')
        if i+1 != 10 :
            output += '\t\t' + str(i+1) + " "
        else :
            output += '\t\t' + str(i+1)
        for l in board2[i] :
            if (l  not in ['-', 'X' , 'O']) and isfinal:
                output += '-' + " "
            else :
                output += l + " "
        output = output.rstrip(' ')
        output += '\n'
    output += '\n'
    return output
```

Showboard Function:

Shows the board. Takes the move player and turn.It takes strings from board and situation function.

```
def showboard(move,player,turn) :
    output = ""
    output+=(f"{player}'s Move\n\nRound : {turn}\t\t\t\t\tGrid Size: 10x10\n\n")
    output+=(f"Player1's Hidden Board\t\tPlayer2's Hidden Board\n")
    output+= board(True)
    output += situation()
    output += f"Enter your Move :{move[0]},{ReversedLetterdict[move[1]]}\n"
    write(output)
```

Situation Function:

Situation function writes the situation of the game which says how many ships are on the board. Remaining_ships_string is used for this function. And this string changes with the move function.

```
def situation():
    output =
f"{Remaining_ships_string['Player1']['C']}\t\t\t\t{Remaining_ships_string['Player2']['C']}\n"
    output += f"{Remaining_ships_string['Player1']['B']}
\t\t\t{Remaining_ships_string['Player2']['B']}\n"
    output +=
f"{Remaining_ships_string['Player1']['D']}\t\t\t\t{Remaining_ships_string['Player2']['D']}\n"
    output +=
f"{Remaining_ships_string['Player1']['S']}\t\t\t\t{Remaining_ships_string['Player2']['S']}\n"
    output += f"{Remaining_ships_string['Player1']['P']}
\t\t{Remaining_ships_string['Player2']['P']}\n\n"
    return output
```

Move Function :

PlayerBoard is where all ship's locations is stored. If move is not equal to '-' which is nothing it changes to 'x' which means player shot it. And checks for if it is sunk or not. Boards is a dictionary and explained before. Length of it is total number of ships if move equals to a location in the list (which has to be). It removes from the list and checks if the length of that list equals to 1(which is the char). If it is true it replace the remaining_ships_string with the X for the sunken ship type.

```
def move(move,player,turn,otherplayer):
    global Remaining_ships_string
    showboard(move,otherplayer,turn)
    if Playerboard[player][move[0]-1][move[1]-1] != '-' :
        Playerboard[player][move[0]-1][move[1]-1] = 'X'
        for i in range(len(Boards[player])):
            for j in Boards[player][i]:
                if move == j :
                    Boards[player][i].remove(move)
                    if len(Boards[player][i]) == 1 :

                        char = Boards[player][i][0]
                        write(f"A {ShipsLetters[char]} has been sunk.\n")
                                Remaining_ships_string[player][char] =
                                …Remaining_ships_string[player][char].replace('-','X', 1)
        return
    else :
        Playerboard[player][move[0]-1][move[1]-1] = 'O'
        return
```

Checkwin function :

Checkwin function works with the remaining_ships_string. In the string there are ‘-‘s and ‘X’s. If there is no ‘-‘ in the string it means that player has lost. It uses this logic and with the if statements it calculates which player wins.

```
def checkwin():
    player1wincondition, player2windcondition = False , False
    for values in Remaining_ships_string['Player1'].values():
        if '-' in values :
            player1wincondition = True
    for values in Remaining_ships_string['Player2'].values():
        if '-' in values :
            player2windcondition = True
    if player1wincondition and not(player2windcondition):
        write('Player1 Wins!\n')
        return False
    elif player2windcondition and not(player1wincondition):
        write('Player2 Wins!\n')
        return False
    elif not(player1wincondition and player2windcondition):
        write("It's a draw\n")
        return False
    else :
        return True
```

Main Function :
Part1 :

This part reads files, handle exceptions and writes kaboom and quits.(if there is a problem) And then with the help of the functions above it stores all of the data in the lists and dictionaries

```
try:
    txtfiles_list= [player1board_txt, player2board_txt, player1moves_in, player2moves_in]  =
sys.argv[1:5]
    txtfiles_paths = [os.path.join(curdir, file) for file in txtfiles_list ]
    txtdict= dict(zip(txtfiles_paths,txtfiles_list))
    IOErrorstring = ""

    for path in txtfiles_paths :
        try :
            with open(path, 'r') as f :
                f.read()
        except IOError:
            IOErrorstring += (txtdict[path]) + " "
    if IOErrorstring != "":
        write(f"IOError: inputfile(s) {IOErrorstring} is/are not reachable.")
        raise IOError
except IOError :
    pass
except IndexError :
    write('kaBOOM: run for your life!')
else:
    try:
        with open(txtfiles_paths[0], 'r') as f:
            x = [lines.strip('\t').split(';') for lines in f.read().splitlines()]
            Boards['Player1']  = find_locations(sortingboard(x.copy()),'Player1')

        with open(txtfiles_paths[1], 'r') as f:
            x = [lines.strip('\t').split(';') for lines in f.read().splitlines()]
            Boards['Player2']  = find_locations(sortingboard(x.copy()),'Player2')

        if len(Boards['Player1']) != 9 or len(Boards['Player2']) != 9 :
            raise IndexError
        with open(txtfiles_paths[2], 'r') as f :
            player1moves = f.read().split(';')
        with open(txtfiles_paths[3], 'r') as f :
            player2moves = f.read().split(';')
```

Part 2:

Main part of the code it is while loop. 2players moves and checks if game is over or not. If not condition stays true and game continues. Out of turn makes it draw(because it will be IndexError.)Last information uses the functions with the False. (It means boards will not be hidden)

```
    condition = True
    i = 0
    while condition :
        try :
            i += 1
            showboard('Player1',i)
            move(*sortmove(player1moves, 'Player2'),i,'Player1')
            showboard('Player2',i)
            move(*sortmove(player2moves,'Player1'),i,'Player2')

            condition = checkwin()

        except IndexError:
            write("Out of turn, It's a draw\n")
            condition = False

    lastinformation = f"Final Information\n\nPlayer1's Board\t\t\t\tPlayer2's Board\n"
    lastinformation+= board(False) + situation()
    write(lastinformation.rstrip('\n\n\n'))
except IndexError :
    write('kaBOOM: run for your life!')
```

User Catalogue

The game needs 4 different txt files for playing. 2 of them indicates boards and others are moves. In the board files ‘;(semicolon)’ stands for the interval of the squares and it should be 9 of them on each line. 10 lines must occur in the board’s text files. The number and the size of the ships are important. Also the letter you use for ship types should be capitalized and correct. These warnings are fatally important and may kaBOOM your game.

You can be rigorous about the number of moves but it is optional. Game continues no matter what you type in the moves txt files until it finds a correct move. So if you want to finish the game before you out of moves , you need to be careful about the moves. An example of the moves txt is like this. 1,B;2,B . So comma splits the move by column and row and semicolon splits the moves.

All errors and information will be in the Battleship.out file. You can examine the progress of the game with this file.