BBM103 FALL 2022

ASSIGNMENT 4 : BATTLE OF SHIPS

2210765020 - SÜMEYRA KOÇ

January 3, 2023

NOTIFICATION : I will use the pronoun "we" in most places throughout the report, so I declare that I did the homework alone, so as not to create the impression that we did the homework in a group.

ANALAYSING

In Assignment 4, we were asked to write a simple Battle of Ship game code. The game will be played on a table of 10x10 , and players place their ships on a table of 10x10 using ";" signs and letters indicating the type of ship in an input file that they will enter when running the game for the first time. The other two input files entered during the first run of the game are the files containing the moves of each of the players (We have used the expression "running the game for the first time" twice, obviously the game is run only once, so it is worth noting that the game is not run more than once.). After the players have entered all the files in full and in the correct order, the game announces the winner and loser or the draw.

– How is the game played, according to what is it lost or won or draw?

Although there are many variants of the Battle of Ship game, we have written the code of a game that can be played with the most basic rules in the Assignment 4 assignment.

In the game, each player has a separate sea (symbolized as 10x10 tables). Before starting the game, players place their navies consisting of 5 types of ships and a total of 9 ships in their seas in such a way that they are horizontal and vertical, and the ships do not overlap (it is already impossible to overlap). Of course, players are unaware of the location of the opposing player's navies and of course the locations of the navies cannot be changed after the start of the game. The aim of the game is to determine the location of the hidden navies of the opposing player before the opposing player does, by making a random hit (so throw a bomb) in each hand. Then each player performs a hit on the desired square in the sea of the opposing player (10x10 table). If a piece of an opposing player's ship is included in the square he hits (a unit of the 10x10 table), that move becomes a "hit", if it is not included, that move becomes a "miss". If a piece of an opposing player's ship is included in the square he hits (a unit of the 10x10 table), that move becomes a "hit", if it is not included, that move becomes a "missing". Each "hit" destroys one unit of a ship, while each "missing" is an unnecessary bomb thrown into the sea. The total number of units owned by the ship varies depending on the type of ship. If an opposing player has detonated (hit) all the units of a player's ship, that ship falls out of the player's navy.

Our game program, which we have written, shows how much the player has left in his hand from which ship type in each round. The player who explodes all of the opposing player's ships before the opposing player wins the competition. So how can the game end in a draw ? It doesn't seem possible from here. As follows: in a round, first the first player and then the second player shoot and move on to the second round. And also the game does not end until a round is completed. If the first player wins, the second player is always given the right because the round has not finished yet. And if the first player has only one ship piece left in his navy, the second player has a chance to end the game in a draw if he hits that particular unit.

– How will the players play the game ?

Unfortunately, in our Assignment 4 assignment, players cannot play the game in such a way that they take turns shooting (that is, we mean that when making a move, they cannot see during the game whether that move has hit or where the opposing player's shots have hit). Players can only enter a file in which they have entered their moves into our program and only see the winner, loser or draw status. Of course, our program (you will also see on the following pages) writes the information of each round to the output file. But for this reason, you will appreciate that the game is not very exciting.

–What kind of problems are there ?

We have explained the game to you under two subheadings, and now let's examine what problems faced us when writing the game's code.

1. First, we need to convert the files that players have placed their navies into data that we can use in our code and "store" them in an appropriate format.
2. Then, in the same way, we have to convert the players' move files into data that we can use effectively in our code and store this data effectively.
3. Player's moves, the moves, pulling the data structures that we put in order, (for each player "player1" and "player2" as separate data structures) "hit" or "missing", we must determine whether they are (of course we have suffered the opposing player's fleet according to the information from the data structure).We need to update the corresponding data structure and determined according to the result.

4. Another problem that we encounter later is to print the information in the data structures we have to the output file in the desired order. the information we will print:
    a. which player has the turn
    b. current number of rounds
    c. ground size
    d. The "hidden boards" of the first player and the second player in turn side by side
    e. under the "hidden board" of each player, the current ship information available in their navy
    f. the move entered by the player whose turn it is
5. We should continue the game until there is no winning, losing or draw situation (even if the error comes out, we should catch the error and except and continue the game)
6. At the end of each round (i.e. after the second player has made his move), we must check whether winning, losing or draw situations have occurred.,
7. If there is a certain result (win or draw), we should declare it and end the program.

Above, we have roughly summarized the problems we are facing. In the Design section, we will talk about how we dealt with these problems.

DESIGN

Our code Assignment4.py as you can see from the "#" shaped comment lines in the file, we have divided it into 6 different sections. The titles of the sections are as follows:

1. # let's check if entered files are proper : In this section we are trying to catch possible errors that may occur when players enter files (the number of entered files may be missing, the entered files may not be opened)
2. # let's start to our code !!!! In this title, we announce that we have started writing our main code.
3. # lets prepare our navies : In this subsection we store the information of the players' navies in the corresponding data structures.
4. # a few additional functions : In this section, there are functions that we need when writing the code of the game and then we write these functions to solve the need.
5. # now it is time for playing : In this section, we play the players' moves one by one after all the arrangements necessary for the game to be played are completed.
6. # finish the game : The part where we check whether the players' win or draw situations occur after each round is completed. If a win or draw situation has occurred, we will finish the game and the program in this section.
7. The last line of our code- it can't be called a chapter because it's a “one-line” code- battleship.this is the line of code that we close our out file.

Above, we briefly mentioned what a total of 6 sections in our program do. Now we will explain the main code in each section and we will explain in detail what problems we encountered while writing this code and how we solved them.

Before we start explaining the sections, let's talk about the lines of code that are not included in any section and are located in the first three lines of the program. First; as you know, we can withdraw the file names that entered in the terminal from the terminal with the sys module. That's why our first line of code is "import sys". The next two lines are related to the output file, i.e. "Battleship.out". In order for our output file to start each new game special and clean, we need to delete the "information left over from the previous game" in the output

file before each new game. To address this need, we have found the following two-line solution :

```
with open("Battleship.out","w") as out_file:
    out_file.write("")
```

We open the file in "w" mode and close it without typing anything green, and all the information in it is thus deleted. Now we can explain the sections.

1# let's check if entered files are proper

I will not share the code because the code for this section is simple. I took the entered files into a list using the “sys.argv” method and entered the list into the for loop and checked whether the file is openable in the for loop. I have printed the non-working files to both the output file and the terminal by specifying their names. Also I checked to see if the number of files entered was correct. If no problems occurred, I equalized the "oyun" variable to "True" and let the rest of the code start working, if any problems occurred, I equalized my "game" variable to "False" and prevented the rest of the code from working. So if an error occurs in one of the files, my code prints the error and terminates the program.

2# let's start to our code !!!!

- The if expression, which includes all the rest of the program except the first part, is in this section. So the first code of this section is the “if oyun :” statement, which also runs all the remaining sections in it. You can think of this expression as a gateway to the rest of the code and the “oyun” variable as a key. In other words, if there are no errors in the first section, the "oyun" variable will be True, and this if expression (i.e. the entire program) will be processed.
- Secondly, in this section, we have prepared "10x10" nested lists of two players called "main_list_player 1 (or 2)", which we call "main list", using the list comprehension method.

  The code written for player1: (the same for player2)

```
main_list_player1= [["-", "-", "-", "-", "-", "-",
"-", "-", "-", "-"] for y in range(10)]
```

Let me explain the logic of main lists: The outermost list has 10 list elements.(each element represents rows in a 10x10 table) Each element in a list type also contains 10 elements.

These elements are just "-" for now (Because in the third part, which we will go through in a moment, we will make changes to this list by pulling data from files with the txt extension in which players set up their navies. By determining the type of the entered ship to be "P", "S", "C", "D" or "B", we will place it in the relevant index of the relevant element of the main list according to the information we have obtained from the file with the txt extension.) And each element inside a list element, in turn, specifies one column. (Info: In the tables written to the output file, there are numbers at the beginning of the rows and letters at the beginning of the columns.) Thus, for both players, we get a data store where we can store the data in each unit of our 10x10 table. In order to be a little more understandable, let me give an example of what we have described above:

Main list is like this = [[...], [...], […], [...], [...], [...], [...], [...], [...], [...]] It has 10 elements. Now let's take a closer look at one element

Main list' one element is like this = ["-", "-", "-", "-", "-", "-", "-", "-", "-", "-"] It has 10 elements.

For example, let's look at what the sixth element in the third element of the list above represents: We said that each of the elements of the outermost list represents a row. The third element represents the third row on a 10x10 grid. Then let's look at the sixth element of the third element of the list, which represents the sixth column to us. Thus, our example positions the third row in the sixth column. Since each column is represented by letters and each row by numbers, our example represents the unit of the 10x10 table (3, F).

- Thirdly, in this section I have a print function called "printt", where I print the outputs both to the terminal and to the output file. This function works with 4 parameter inputs. `def printt(p, e, hangi1, hangi2):`

  First of all, I would like to remind you that the tables that we print at the end of the game (when the win situation or draw situation occurs, or when the players' moves are over) and the tables that we print during the game (when the moves are made) are not the same. the tables and sentences printed during and at the end of the game are as follows, respectively :

```
Player1's Move

Round : 39                  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 O - - O - O X - O O       1 - - - - - - - - - -
2 - - O - - - X - O -       2 - - X - - X X O O X
3 - - - - - O - - - -       3 - - - - - - O - - X
4 O - - - - - X O - O       4 X O X - O O O - - -
5 O O O - - O - - - O       5 - - - - O O X - - -
6 - X X X X - O - O -       6 O - O - - - O O - -
7 O O - - - - - - - O       7 O - - O - - - - - -
8 O - - O O O - - O X       8 - - - - - - O O X O -
9 - - - - - - O - - -       9 - - O - X - - - - -
10- - - O O - - - O -       100 - O O O - O O O -

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

Enter your move: 3,E
```

Fig1

```
Player2 Wins!

Final Information

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

Fig2

Differences from each other are these :

for what is printed during the game:

1. It shows who has the turn of the game,
2. the current number of rounds and the grid are shown to you,
3. the ships of both players are hidden, that is, the locations are not shown,
4. It shows how much of which ship type is left for two players,
5. the moves entered by the player whose turn is shown

for what is printed end of the game:

1. It is printed who won the game or whether it ended in a draw,
2. the locations of the remaining navies of the players are shown

Because of these differences, I wanted to be able to print two separate states in one print function instead of writing two separate print functions. The parameter "p" gives which player has the turn. the parameter "e" gives the move of the player whose turn it is. The "hangi1" and "hangi2" parameters give lists of data to be printed. (Due to the differences I described above, the function will print different editions according to the states of the "hangi1" and "hangi2" parameters.)

```
if hangi1 == player_1_hidden_board and hangi2 ==
player_2_hidden_board and draw_or_not == False:
```

The "if" expression above is an expression in which the program decides which of the two table types we described above to print. If the lists entered as parameters when calling the

function (which1 and which2) are equal to the hidden lists, it means that the function was called during the game, if it is not equal, it means that the function was called at the end of the game. And according to these if expressions, the paths followed by the program in the printt function will be different.

It's going to be a little complicated, but what's the “draw_or_not” variable ? If the draw_or_not parameter is equal to True, it indicates that the match is continuing, if it is equal to False, it indicates that the match ended in a "draw". When the game ends in a draw, we call the printt function to print the tables by equalizing draw_or_not to False.So what does this provide us with? If it were not for this expression (that is, if we could not tell the printt function that the match ended in a draw), the game ended (and it ended in a draw), our program would confuse the type of table it would print and print the tables it printed during the game. (the reason is based on hangi1 and hangi2 parameters, but it is really impossible to tell how it happened in writing.)

To sum up, there are a total of two if expressions in the form we shared above, and thanks to these if expressions, we can print two different types of output to the printt function.

Apart from that, another problem with the printt function is to convert the ship information in the players' navies into writable data in some form and print them to the output file and terminal. For this, I wrote the "information" function, which we will describe in the fourth section. When this information function is called, it returns the information that we can directly write next to the ship type. The parts about the printt function other than those described above are hardly worth mentioning. That's why I'm finishing this chapter and moving on to the third chapter.

3# let's prepare our navies

Because in the third part, which we will go through in a moment, we will make changes to this list by pulling data from files with the txt extension, where players build their navies. By determining the type of the entered ship to be "P", "S", "C", "D" or "B", we will place it in the relevant index of the relevant element of the main list according to the information we have obtained from the file with the txt extension. In order not to tell the same things once again, I am sharing the third part function understanding mentioned a little in the second part again :

(Because in the third part, which we will go through in a moment, we will make changes to this list by pulling data from files with the txt extension in which players set up their navies. By determining the type of the entered ship to be "P", "S", "C", "D" or "B", we will place it in the relevant index of the relevant element of the main list according to the information we have obtained from the file with the txt extension.)

In addition, by writing this section inside the "try:" block, we prevent the rest of the code from working by equating our "oyun" variable to False in case of any errors. And we are printing an error message.

4# a few additional functions

There are two functions written in this section to meet the needs. One is called "list_pop" and the other is called "information" functions. If you remember, we mentioned the information function in the third item of the second section, but it was not detailed. So what do these functions do?

- `def information(h, p):`

  This function is a function that returns which player has how much of which ship type he has left in his hand with the signs "-" (for ships that still exist) and "x" (for exploding ships). the main difficulty I had while writing this code and the part that bothered me was the information of the ship types, the number of which was more than one. In order to avoid confusion here, we asked players to share the locations of the ship types (B and P), the number of which is more than one (B1:6,B;right/down; written in the following patterns) in the txt file. The names of the files have to be fixed, OptionalPlayer1.txt and optionalplayer2.txt, and have to be in the same folder as the file with the py extension where the code is running. These optional files have made our work much easier. Let's take a look at the functioning of the function a little

  This function is a function that works with two parameters. The parameter "p" gives which player is asked for navy information, and the parameter "h" gives the type of ship whose information is requested. In other words, this function returns only one ship type information in each operation.(by inserting the "ship_list" in the printt function into the for loop, we will thus get the information of all ships (ship_list=["Carrier", "Battleship", "Destroyer", "Submarine", "Patrol Boat"] after

inserting it into the for loop, we will get the initials of the ship names as the "h" parameter.)

As we mentioned at the beginning, the difficult part is the part of calculating the information of ship types, the number of which is more than one. We find the information of the ship types, the number of which is one, with a simple if expression and a few lines of code. We obtain the information of the ship types, the number of which is more than one, by using the optional files. To review the code, see the programmer's catalog. Another point that I consider important for this function is the part where we use the parameter "p". I used the logic in this part in most parts of the entire code, I can say that it saved me from a serious workload. The fact is that our program is a program that does the same things for two players with different data. If we wrote different codes for each player, there would be twice as many lines of code as the current total number of lines of code (the current number of lines of code is 326). That's why the functions or pieces of code in our program are designed to do the same job for two different players using parameters or if expressions. I'm sharing the code structure I'm trying to explain :

```
def information(h, p):
    if p == 1:
        dosya = "OptionalPlayer1.txt"
        liste = main_list_player1
    elif p == 2:
        dosya = "OptionalPlayer2.txt"
        liste = main_list_player2
```

After that, I comfortably use my variables (dosya, liste) in my code, to which I assign different values according to the entered parameter. I have used this code logic in most parts of the entire code.

- `def list_pop(k):`

  It would be more understandable if we described this function in the fifth chapter, but it would be more appropriate to mention it now. The fact is that there is a very important variable in our code called "counter". This variable sees more than one of our jobs, one of which is to tell which index move will be removed from the players' "move list" according to the number of rounds (we will mention in the fifth chapter what the move list is, but you have understood what it means). In other words, the

goal is to ensure that each player makes the same index move in the round. The problem that caused us to write this function arises here: if a player's move has errors such as "index error" or "value error", that move becomes an unworkable move. And the turn must be on that player again and move on to another move in the player's move list. While doing this, we need to keep the counter stable so as not to skip any moves of the other player. We found the solution to this problem by removing the element that gives error from the move list.

```
def list_pop(k):
    if k == 1:
        moves_list1.pop(counter)
    elif k == 2:
        moves_list2.pop(counter)
```

As you can see above, we are choosing which element to remove from the list according to the entered parameter. The error; because it appears in the counter'th index, we remove that element directly from the list by typing .pop(counter).

So we have finished this section as well, we can move on to the fifth section

5# now it is time for playing

- My first code in this section is to store the data in the move files I receive from users in the move lists.

```
with open(sys.argv[3], "r") as file:
    readed1 = file.read()
    readed1 = readed1.rstrip(";")
    moves_list1 = readed1.split(";")
with open(sys.argv[4], "r")as file:
    readed2 = file.read()
    readed2 = readed2.rstrip(";")
    moves_list2 = readed2.split(";")
```

- We mentioned the list of moves in the second item of the fourth section. Now you understand what happened. We store the move data of player1 and player2 in the “moves_list1” and “moves_list2” data stores (type list), respectively.
  Then we define the counter variable, which is a very important part of our code, for the first time here.

```
counter = 0
```

- After this code, our function called "move" is written, which is the function where the players' moves are played.

```
def move(a):
```

As you can see, this function works with one parameter. the parameter “a” gives the player who made the move. when calling the function, 1 or 2 is written instead of a.
The following 3 lines of code call 3 variables (which is what two of these variables are, I will explain in the next part) from global.

```
global error_type
global sira_kimde
global counter
```

After these lines of code, I once again applied the code logic that I mentioned in the first article of the fourth chapter and said, "I used this code logic in many parts of the program."

```
if a == 1:
    list = moves_list1
    main_list = main_list_player2
    player_hidden_board = player_2_hidden_board
elif a == 2:
    list = moves_list2
    main_list = main_list_player1
    player_hidden_board = player_1_hidden_board
```

This time I used this code logic to assign different values to the variables "list", "main_list", "player_hidden_board" according to the value of the parameter “a”. The “list” variable gives the players' move lists, the “main_list variable” gives the main

list where the players' navies have information, the players', and the "player_hidden_board" variable gives the list whose data will be printed during the game (i.e., these are lists where the location of their navies is unknown, these hidden_lists).

In this way, I will be able to run the function in the same way for the player I want by using only these 3 variables in the function.

Thus, instead of writing the same function twice differently for two players, I made sure that a single function could work for two players. Thanks to this code logic. I state again: I have used this code logic in most parts of the program.

Then I write and call the printt function with the corresponding parameters into it and print the current information to the output file and terminal. (As I shared as Fig1 above)

```
printt(a, list[counter], player_1_hidden_board,
player_2_hidden_board)
```

Info: I am pulling the related move directly from the move list by typing "list[counter]" in the parameter section named "e", which is the move parameter of my printt function.

If we continue to progress inside the move function, control codes will appear, in these control codes I check with if statements whether the player's move entered in that round could be incorrect. If an incorrect situation occurs, I raise Index error or Value error or Assertion error (these errors I raise will be captured by except later, I will mention in the next item of this section).

It's time to explain the error_type variable. We have three different types of assertion error. In order to separate these types from each other and print different errors on all of them, if an assertion error occurs, we equate the "error_type" change to the type of the corresponding error (1, 2, or 3) before raising the assertion error. For example:

```
if 75 <= ord(letter) <= 90:
    error_type = 2
    raise AssertionError
```

If the above error type appears, that is, if the ord (letter) number is less than 75 or greater than 90, we have equalized our error type to two.

t

Then, the except Assertion statement captures this error (which is Assertion Error)

```
except AssertionError:
```

It will be explained in the other article where the errors were caught.

Let's share the continuation of the code we shared above :

```
except AssertionError:
   if error_type == 3:
. . . . . .
   elif error_type == 2 or error_type == 1:
. . . . . .
   else:  # so error_type ==1
. . . . . .
```

We learn what the error_type variable is with the if elif expressions. Then we write different codes under the if elif expressions according to the type. Let's remind you, except we'll see the other parts in the article.

Another issue that we consider worth mentioning about the move function: Checking the element corresponding to the player's move entered in the opponent's main data store, if this element is equal to "-", we accept the entered move as "missing" and replace the corresponding element with the letter "O". If this element is equal to any of the letters "P","S","C","D","B", we accept the shot as a "hit" and replace this element with the letter "X". Let's share code :

```
if main_list[number - 1][ord(letter) - 65] == "-":/
   player_hidden_board[number - 1][ord(letter) - 65] =/
"O"
   main_list[number - 1][ord(letter) - 65] = "O"
elif main_list[number - 1][ord(letter) - 65] == "C" or/
"D" or "B" or "P" or "S"  # I shortened this place to keep the code short, in fact, these "or" statements do not give results in this way. But you understand the logic.
   player_hidden_board[number - 1][ord(letter) - 65] =
"X"
   main_list[number - 1][ord(letter) - 65] = "X"
```

As you can understand from the code, we are changing both the "player_hidden_boards" and the "main_lists" that we will use for printing during the game. (we will print these main lists at the end of the game under the title “ Final Information ”.)

The last piece of code of the move function is the following :

```
if a == 1:
    sira_kimde = 2
elif a == 2:
    sira_kimde = 1
```

We have opened the function of this line of code as a comment line in our code. We have said that we will describe the variable "sira_kimde" in the other item. Now we can tell it by the way, because it will be more comfortable for you to understand when it is explained in the other article.

That is: if an error occurs in the moves entered by the player, we will have told our code which player the error occurred in with this "sira_kimde" variable. And what will happen when our code finds out about this? If you remember, we have said that we have removed the incorrect moves from the corresponding player's move list in the list_pop function and kept the counter constant. It is thanks to this variable that we succeed in selecting elements from the correct player's move list.

In turn, we don't just use your device in this situation, there are many parts that we use ourselves, and it helps us a lot. In addition, we do not solve the problem we described above only with the sira_kimde variable. We can also use a variable called num (which we will talk about later).

In summary, since the corresponding player's move is made in the above code fragment, the sira_kimde variable is equalized to 2 if it is 1, or it is equalized to 1 if it is 2.

Let's move on to the last item of the fourth section

- In the third article of the second section, while talking about the printt function, we mentioned the utility of the draw_or_not variable. Here we will define that

draw_or_not variable in this article. Even this identification is the first code of this article :

```
draw_or_not = False
```

The next code from this code is the code in which we define the sira_kimde_ variable (the variable we mentioned in the previous article)

```
sira_kimde = 1
```

We open the while loop to start playing the game later. SAll the codes that we will describe later (6. including the section) will be in this while cycle. Do not forget about it :

```
while oyun:
```

We explained in the first chapter what the game variable is. After this line of code, there are two lines of code :

```
num = 0
error_type = 0
```

We described the error_type variable in the previous article, that is, in the third article. This is where that variable was first defined.

The “num” variable is very important and is a variable that we use in most parts of the program. Its function is very similar to the function of the variable “sira_kimde”. gives which player the code is currently working for. There is a “try:” block after these lines :

```
try:
```

In this try block, I check to see if there is a move that player can make in the move list according to which player's turn is first, if there is no move that player can make, I call the printt function to print the tables in the form of Fig2 mentioned above and I remove the program from the while loop using the break expression.

```
    if sira_kimde == 1:
        if counter > len(moves_list1):
            . . . .
            printt(1, "2,E", main_list_player1,/
main_list_player2)
            break
```

If a player has a move to make, I call the move function by typing the corresponding player's number (1 or 2). (i explained it in the third article)

```
        num = 1
        move(1)
```

Since we wrote num == 1 before calling the move function (because it's time to player1), our code will be able to easily see which player the error originated from in a possible error inside the move function.

I apply the same process above to the second player as follows :

```
    elif sira_kimde == 2:
        . . . . . .
        roundd += 1
        counter += 1
```

Unlike player1 's block, the reason we have included the two codes above in this statement is this: As we mentioned in the second section, each round is completed after the second player makes his move. That is the reason why we put these two codes inside the Player2's block. That is, the “counter” and “roundd” values (the roundd variable gives the number of rounds) are increased only after the second player has "successfully" made his move.(We've mentioned it before, but let's talk about it again. If there is an error in the moves, except the blocks will catch them, and since we are in a while loop and we have not increased the counter and round numbers, it will be the second player's turn again. We used the same logic for the first player.)

Thus, we exit the Try block and enter the except blocks.

- In this article, we will process 4 different types of except blocks.

```
except IndexError:
      . . . . . .
except ValueError:
      . . . . . .
except AssertionError:
      . . . . . .
except:
      . . . . . .
```

We do not need to explain each except block separately. The operations that blocks generally perform are as follows: Removing the wrong move from the move list with the pop_list function, printing warning messages to the terminal and to the file, exiting using the break expression if it is necessary to exit the while loop.

Chapter six is over, we can move on to chapter seven. Remember that we are still in the while loop.

6# finish the game

In this section, we check whether win or draw situations occur in the game at the end of each round. If such a situation has occurred, we call the printt function to print tables in the form of tables of Fig2 (we mentioned what we meant before) and exit the while loop using the break expression.

This section (i.e. the last part of the entire code) is written inside the following if statement :

```
if sira_kimde == 1:
```

The sira_kimde variable also makes our job easier here, giving us which player is the player who has not made his move, but will make it. In the above expression, you saw that the sira_kimde variable must be equal to the number 1 to check whether the game has been concluded or not. The reason for this (we have emphasized only in previous sections) is that the game will not end without the second player making his move. In other words, the game cannot end before the round is completed. As a result of this rule, we have written the if

statement above in that way. The piece of code that follows the above line of code is as follows :

```
set1 = set()
set2 = set()
for x in range(10):
    for y in main_list_player1[x]:
        set1.add(y)
if "C" in set1 or "B" in set1 or "D" in set1 or "S" in set1/
or "P" in set1:
    player1_station = True
else:
    player1_station = False
for x in range(10):
    for y in main_list_player2[x]:
        set2.add(y)
if "C" in set2 or "B" in set2 or "D" in set2 or "S" in set2 or
"P" in set2:
    player2_station = True
else:
    player2_station = False
```

I created the set1 set for player1, I created the set2 set for player2. Then I put the main lists of the players in the for loop and placed all the element types in it ("-", "X", "O", "P","S", "D", "B","C") in the corresponding set data store (set1 for player1, set2 for player2). Then I checked to see if there were any ship parts in these set data stores I created. If there are any ship parts, the player_station variable (according to the respective player: player1_satation for player1, player2_station for player2) is equal to True, if there are none, it is equal to False.

When you share the continuation of the code I shared above, you will see what the player_station variables do.

```
if player2_station == False and player1_station == True:
     . . . . . .
elif player2_station == True and player1_station == False:
     . . . . . .
elif player2_station == True and player1_station == True:
     continue
else:
     draw_or_not = True
     . . . . . .
```

In the first if block, there is a winner player1 status, in the next elif block there is a winner player2 status, in the next elif block there is no winner status, (that is, the game continues, there is only a continuous expression in that part) in the next else section there is a match ends in a draw status.

The things we do in the blocks where win or draw situations occur are the same. The printt function is called to print from tables in the form of Fig2. and exit the while loop using the break expression. You can see the detailed codes in the Programmer's Catalog.

We also consider it important to emphasize the following: the block where the match ended in a draw (i.e. in the else: block) has the expression draw_or_not = True. In other words, when we call the printt function, our program will know that the match ended in a draw, and as we mentioned about this topic when describing the printt function, which we described in the 3rd article of the second part, our program will not confuse the lists and thus print the tables that are requested to print.

So we have explained all the chapters. finally, we closed our output file with the expression

```
out_file.close()
```

in the last line of our code.

PROGRAMMER'S CATALOG

Before starting this "Assignment 4", I downloaded the Battle of ships game called "Fleet Battle" to my phone to understand the logic of the game and to create the game's code sketchily in my head, and I got the habit of playing as I got bored and thinking about the code I would write while playing. It took me about half an hour to analyze the Assigment4.pdf. Then I spent my free time thinking about how to write the code and jotted down all the ideas that came to my mind in a notebook on my phone.

Although I am not yet at a sufficient level to give advice on programming, I have had enough homework to make observations and experiments and generate ideas in them. One of the ideas I have come up with is that, after understanding the problem that needs to be solved, instead of immediately turning on the computer and trying to write the code for the problem, poke at that problem as it comes to your mind during the day (thinking abstractly about the following questions: What are the subproblems of the problem? Which part causes me the most difficulty? How do I solve them ? What will be the logic of my main code ?) It will allow you to approach the problem "in its entirety" and write a very clean code in a very short time after a few days of brain training.

I spent a few days thinking about how I should write my code as I analyzed it above, the next day I created the main structure of my code (I divided it into the sections I mentioned in the Design section, I built the structures of the necessary functions), and then in the remaining days I dealt with small problems (they are small, but they are too many, so their solutions take a lot of time). I have implanted the latest error capture code fragments to various parts of my program and made the final checks. After this process, my program was running without errors (I deliberately tested my error capture codes by entering an incorrect file and entering incorrect moves.). I finished the report part in a total of 6 hours by spreading it over three days.
I have explained the entire code in the Design section, so I will share the functions we have written in this catalog. Before moving on to fonskions, I realized that my code is very flexible and

I would like to say that the variables I have assigned are in place and are very useful for processing. All kinds of new features can be easily added to my code or an unwanted feature can be easily removed.

I repeat, I have processed all the functions in detail in the Design section, so I consider it unnecessary to describe the same things here.

1- printt(p, e, hangi1, hangi2) function :

```
def printt(p, e, hangi1, hangi2):
    if hangi1 == player_1_hidden_board and hangi2 ==
player_2_hidden_board and draw_or_not == False:
        print(f"Player{p}'s Move\n")
        out_file.write(f"Player{p}'s Move\n\n")
        print(f"Round :{roundd:<21}\tGrid Size: 10x10\n")
        out_file.write(f"Round :{roundd:<21}\tGrid Size:
10x10\n\n")
        print(f"Player1's Hidden Board\t\tPlayer2's Hidden
Board")
        out_file.write(f"Player1's Hidden Board\tPlayer2's
Hidden Board\n")
    alp_list = [chr(y + 65) for y in range(10)]
    b = " ".join(alp_list)
    print(f"{b:>21}\t\t{b:>21}")
    out_file.write(f"{b:>21}\t\t{b:>21}\n")
    for x in range(10):
        a = " ".join(hangi1[x])
        print(f"{x + 1:<2}" + a, "\t\t", end="")
```

```
        out_file.write(f"{x + 1:<2}" + a+ " \t\t")
        b = " ".join(hangi2[x])
        print(f"{x + 1:<2}" + b)
        out_file.write(f"{x + 1:<2}" + b+"\n")
    if hangi1 == player_1_hidden_board and hangi2 ==
player_2_hidden_board and draw_or_not == False:
        for x in gemi_listesi:
            returnn1 = information(x[0], 1)
            returnn2 = information(x[0], 2)
            print(f"{x:<15}{returnn1}\t\t{x:<15}{returnn2}")
```

```
out_file.write(f"{x:<15}{returnn1:<8}\t\t{x:<15}{returnn2}\n")
        print(f"\nEnter your move: {e}\n")
        out_file.write(f"\nEnter your move: {e}\n\n")
```

2- information(h,p) function :

```
def information(h, p):
    if p == 1:
        dosya = "OptionalPlayer1.txt"
        liste = main_list_player1
    elif p == 2:
        dosya = "OptionalPlayer2.txt"
        liste = main_list_player2
    if h == "C" or h == "S" or h == "D":
        for x in range(10):
            if h in liste[x]:
                return "-"
            elif x == 9:
                return "X"
    else:
        with open(dosya, "r") as optional_file:
            for line in optional_file:
                right_or_down = line.split(";")[1]
                type_of_bottle =
line.split(";")[0].split(":")[0]
                coordinate = line.split(";")[0].split(":")[1]
                if right_or_down == "right":
                    dic[type_of_bottle] =
liste[int(coordinate.split(",")[0]) - 1][

int(ord(coordinate.split(",")[1]) - 65):int(

ord(coordinate.split(",")[1]) - 65) + 2]
                if right_or_down == "down":
                    if type_of_bottle[0] == "P":
                        dic[type_of_bottle] = [
                            liste[int(coordinate.split(",")[0])
- 1][ord(coordinate.split(",")[1]) - 65],

liste[int(coordinate.split(",")[0])][ord(coordinate.split(",")
[1]) - 65]]
                    elif type_of_bottle[0] == "B":
                        dic[type_of_bottle] = [
                            liste[int(coordinate.split(",")[0])
- 1][ord(coordinate.split(",")[1]) - 65],

liste[int(coordinate.split(",")[0])][ord(coordinate.split(",")
[1]) - 65],
                            liste[int(coordinate.split(",")[0])
+ 1][ord(coordinate.split(",")[1]) - 65],
                            liste[int(coordinate.split(",")[0])
+ 2][ord(coordinate.split(",")[1]) - 65]]
        returnn = []
```

```
        for x in dic:
            if h == x[0]:
                if h in dic.get(x):
                    returnn.append("-")
                else:
                    returnn.insert(0, "X")
        return " ".join(returnn)
```

3- list_pop(k) function :

```
def list_pop(k):
    if k == 1:
        moves_list1.pop(counter)
    elif k == 2:
        moves_list2.pop(counter)
```

4- move(a) function :

```
def move(a):
    global error_type
    global sira_kimde
    global counter
    if a == 1:
        list = moves_list1
        main_list = main_list_player2
        player_hidden_board = player_2_hidden_board
    elif a == 2:
        list = moves_list2
        main_list = main_list_player1
        player_hidden_board = player_1_hidden_board
    if 0 < len(list[counter]) < 3:
        raise IndexError
    printt(a, list[counter], player_1_hidden_board,
player_2_hidden_board)
    hamle = list[counter].split(",")
    letter = hamle[1]
    number = int(hamle[0])
    if number > 10:
        error_type = 1
        raise AssertionError
    if hamle[1] not in ord_list or len(hamle) != 2 or
len(hamle[1]) != 1 or ord(letter) > 90 or ord(letter) < 65:
        raise ValueError
    if 75 <= ord(letter) <= 90:
        error_type = 2
        raise AssertionError
```

```
    if main_list[number - 1][ord(letter) - 65] == "-":
        player_hidden_board[number - 1][ord(letter) - 65] = "O"
        main_list[number - 1][ord(letter) - 65] = "O"
    elif main_list[number - 1][ord(letter) - 65] == "C" or
main_list[number - 1][ord(letter) - 65] == "B" or \
            main_list[number - 1][ord(letter) - 65] == "D" or
main_list[number - 1][ord(letter) - 65] == "S" or \
            main_list[number - 1][ord(letter) - 65] == "P":
        player_hidden_board[number - 1][ord(letter) - 65] = "X"
        main_list[number - 1][ord(letter) - 65] = "X"
    else:
        error_type = 3
        raise AssertionError
    if a == 1:
        sira_kimde = 2
    elif a == 2:
        sira_kimde = 1
```

Possible errors that we check in our program, errors that occur other than these are called "kaBOOM: run for your life!" we are printing.

Errors that may occur when entering a file :

1. the number of files entered may be small
2. the entered files may not be able to be opened
3.

Errors that may occur as a result of incorrect moves in the move files:

1. the entered move may be an incorrect statement (moves such as“ "A,1;“ ”1,1;“ ”A,A;“ ”5,E10,G;“ ",A;” "A;” “,;” “;” “1,;” “1;” “11, A;” "5,K;")
2. If the players have no moves left and the game is still not over

USER'S CATALOG

Dear user, first of all, I recommend that you read the section where I explained how to play the game under the title Analaysing and thus get acquainted with the game.

How to play the game :

1. First, place your ships in a txt file in such a way that it looks like this.

   ;;;;;;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

2. Then write the moves you will make in order in your move file in such a way that it looks like this (and it will be a single line):

   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; . . . . .

   (the row of numbers indicates the column of letters)

3. then wait for your opponent to prepare them too

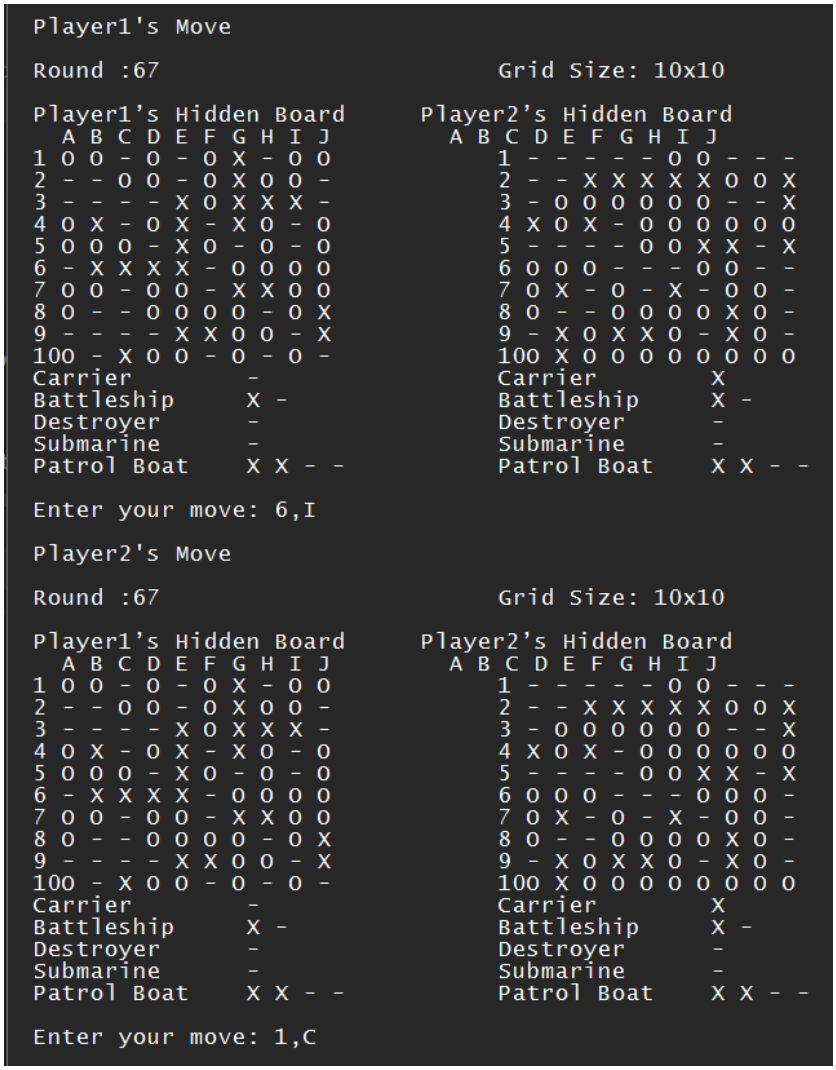
4. then create an empty output file called Battleship.out.

5. then put these total 5 files that you prepared together with your opponent and the python file named "Assignment 4.py" in a folder.

6. then open your terminal and copy the path to the folder containing all the files and move the terminal to that folder with the cd command.

7. then type the next statement in the terminal (you must enter the file names correctly in the parts in quotes or the program will not work): python3 Assignment4.py "your navy file" "the opponent's navy file" "your move file" "opponent's move file"
8. then you can open the "Battleship,out" file and see who won or whether the match ended in a draw. There are articles separated by round numbers in this output file. by scrolling up the file, you can see which of your moves has produced results.

To summarize with an example what the information in the output file is talking about, we : This picture shows that after the first player makes his move, the second player passes the turn and he also makes his move.



The expressions "Player1's move" and "Player2's move" at the beginning indicate which player has the turn. The following round statement indicates which round the game is currently in. Then, the hidden boards (i.e. seas) of player1 and player2 are shown side by side. After that, the "-" marks indicate which ship type each player has on the bottom of his board and how much. Then the move made by the player whose turn it is is shown in the enter your move section. You can follow the game using this information. If there is the last winning player, the winning player is announced and the locations of their navies are shown to the losing player on that player's board.

Enjoyable games, I hope you like it…