HACETTEPE UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

BBM 103: INTRODUCTION TO PROGRAMMING LABORATORY I

ASSIGNMENT 4: BATTLE OF SHIPS

STUDENT NUMBER: 2220356114

NAME: UYGAR

SURNAME: ERSOY

DELIVERY DATE : 03.01.23

Table of Contents

ANALYSIS

In this problem, a strategy and guessing type of game for two players called Battleship, which is also known by different names such as Sea Battle or Battleships, will be designed. Each player places their warships to their respective grids at the start of the game. Other player should not see the placement of the opponent's ships. At total, four of these grids will be used, two for each player. One of them will be used to record the taken shots from opponent and to save the shots to see if opponent successfully hit any part of the warships of the player. Other grid will be used to record the shots that each player took to hopefully make a hit at the opponent ships. Each grid will consists of 10 by 10 squares, 100 in total. Depending on the type of the ship, each of them them will be placed consecutively at pre-decided lengths. For example, Carrier type of warship will be placed either horizontally or vertically, throughout 5 consecutive squares. Each ship will be placed horizontally or vertically. At each square of the 100 squares, each of them possibly will hold only 1 ship part. None of two ships will share a common square between them. Each player will have 1 carrier, 1 destroyer, 1 submarine, 2 battleships, and 4 patrol boat. Each of theirs length will be the same for each player for different types of ships. Carrier will have a length of 5 squares, Battleship will have a length of 4 squares, Destroyer and Submarine will have a length of 3 squares and Patrol Boat will have a length of 2 squares. After placing the warships to the grids, game is ready to start. For each round, starting from Player1, each player will take a shot at the opponent's grid to make a hit. After each shot, if it successfully hit a ship, "X" symbol will be placed to emphasize that a ship has taken a hit. Otherwise, "O" symbol will be placed to show that the shot did not hit any ships. This process will continue accordingly and after each round, each ship will be checked to see if any of them completely sunk by the shots. If a ship sunk, "X" will be placed for respective ship to indicate that ship is sunk. Also, at the end of each round, the game will be checked to see if any of the players has won the game. If any of them has won it, game will stop and program will announce that respective player has won the game. If both of their ships all sunk, the program will announce that the game is a draw. Furthermore, if first player has won the game before round ends, second opponent still use his/her last shot and then the winner will be announced with the final state of each of their grids. The parts of the ships that did not take a hit throughout the game will be displayed at the end of the game with the winner. All this process will be carried out using the ships placements from command line files and attack positions from command line files. Ship positions will be given in a file from command line for each player and using those files ships will be placed on respective grids. Each ship will be represented in those files using their name's first letter as capitalized. After ship placements, pre-decided shots will be read from attack files, which were given as arguments from command line. At each shot, representing square in the grid will be checked to see if a ship has a part in that square or not. Each shot will consists of two parts: first is the column value which ranges between 1 and 10 both inclusive, and column value which ranges from A up to and included I in alphabet (3,A for example). At each round; round number, current player, state of each players' grid will be displayed with letter "X" if a hit occurred in a given square, with letter "O" if a miss happened in a given square and character "-" if no attack happened at a given square in each grid. The game will proceed as mentioned above and at the end of the game, winner will be announced if no draw happens. In that case, a message will be printed to state that a draw happened. In case of any invalid or wrong inputs in given files, testings will be applied to each one of them as well for program safety.

DESIGN

Reading Ship Placements From Input Files

To read the placements of ships for each player, read_file() function will be implemented. This function will receive a file name as a parameter and will return a list that contains each line of the given file. For each player, their secret ship placements will be provided with a file from command line as an argument. These file will contain letters that represent the valid ship types. Letters will be separated using “;” character inside of the file. In each line of the file will have nine “;” characters and ship characters will be placed between these characters to represent their place in the grid. For example, let say the first line of one of the file is given as follow: ;C;;;;;;;;. This means that part of a Carrier will be placed at 1,B. Each file will have 10 lines with this criteria inside them. read_file() function will read this files line by line and will get rid of the new line character at the of each line. For each line, they will be appended to a list that is created at the beginning of the process and that list will be returned to execute further processes.

Placing Ships

To place the ships to grids for each individual player, place_ships() function will be implemented. This function will receive the returned list from read_file() function that contains the ship placements as a parameter and place the ships to grids accordingly. It will create a multi-dimensional Python list that contains 10 lists inside of it. Each individual list will have 10 “-” character initially inside them to represent empty squares of a grid. After that, given list as a parameter will be iterated over and each string inside of that list will be checked to find the ship places. Each string in given list as a parameter will represent a line from the file that contains the placement of ships. While iterating each string, if a character other than “;” was found, this means at that position, a ship has a piece to be placed to the grid. Position will be received as follows, index of the whole string plus 1, that are being checked currently, is the row number of that piece and the number of “;” character inside that string from beginning up to but not included to current character will give the row number. According to this algorithm, each part of every ship will be placed to grid of each player and that grid, multi-dimensional Python list, will be returned.

Reading Shot Positions

Shot positions will be read as the same way ship positions are read. Using read_file() function, a file name will be given as a parameter. Shot positions will be given inside of a file from command line. Total shots for the whole game will be given as a single line inside of that file. Each shot for every round will be separated by “;” character. Every shot will consist of two parts that are separated by a comma; first one to represent the row value (1,2,3,….,10) and second one to represent the column value (A,B,C,….,J). read_file() function will read this file and return it as a list that contains a single element which is a string that holds the shots. Returned list will be separated by “;” character using split(“;”) command to extract each individual shot. This process will result with a list that contains strings that represents each shots for every round for the given player.

Extracting Ship Positions For Ships That Exists More Than One

Battleship and Patrol Boats have more than 1 representative of them in each players' grid, namely, battleship has 2 ships and patrol boats have 4 ships in each grid. To extract their positions in every grid, battleship_and_patrol_positions() function will be implemented. Their positions will be given inside of a file, which exists in the same directory with the program. In each line, first letter of the ship with a number that represents its order will be given with the coordinates of its starting piece in the grid and the direction that it is aligned in the grid. For example, let assume this line is given inside of the file as a line B1:6,B;right;.What this means is that, first battleship is starts at row 6 and column B and continues horizontally at the given length of itself. The structure of each line will be the same as the given example. Only "right" and "down" words will be given to indicate the directions of the ship alignment. This function will be called with two parameters, namely, a list that contains the each line of the mentioned file above and an integer to indicate which player's ships these are. File will be read using read_file() and resultant list will be given as parameter. According to the value of second parameter, position of each ships will be recorded to the lists that are created in the global scope. The list will be iterated over and row and column value according with the name of the ship will be extracted from each line. Row and column value will be added to a list according to the alignment and a boolean True value will be added to this list to indicate that this ship is swimming currently. This will be helpful in the future. Finally, this list will be appended to the list at the global scope to hold the position information.

Playing the Battleship Game

After obtaining the whole necessary information about the placements of ships and placing them to the grid, game is ready to be played, which will happen via play() function. This function will take no arguments. It will read shots of each individual player for every round and according to the result of each shot, it will call other functions to make the informative printing information to both console and to a file. While reading from shots of each player, if it encounters an invalid type of shot, it will print a message about it, and it will continue to read from the same player until it founds a suitable shot to be shot. This will be done using a while loop. Unless function gets a valid input, it will continue to read from same player until it finds a correct input, which breaks the loop and continues with the other player. As long as both of the players have a shot left to take, outer while loop will maintain the process of the game. After each player hit or miss a shot, other grid of the player that is used to save the opponent's shots at them, will be updated accordingly and each ship will be checked if they are still swimming or they are sunk by calling other functions. At the end of each round, a function called is_won() will be called to see if any of the players have won the game or not. If that is the case, game will be terminated and the winner will be announced with the final state of each grid. Otherwise, game will continue with the same rules until one of them wins, draw happens, or one of the players exhaust her/his shots.

Printing Grids After Each Shot

After each player took their shot, using print_table(), current state of the grid of each player will be printed. This function will have 5 parameters: hidden grids that holds the records of each shot for each player (multi-dimensional list), an integer to print the round number, an integer to represent which player has taken the shot, and an integer that gives the index of the shot at the shots list. According with the proper spacing, player information, round number, grid size of the board, and the state of the boards will be printed using this function. It will take each hidden grid of players and print them row by row with their current state. By calling another function, states of each ships will be printed for each player. If any ship is sunk, an “X” will be placed for that ship. After that it will print the players shot using the given parameter.

Checking if Any Player Has Won The Game

After each round, using is_won() function, the game will be checked to see if any player has won it or a draw has occurred. It will take to parameters, namely, hidden grids of each player. By counting the number of “X” in each grid and comparing the total length of each ships in the board, the state of game will be decided. If total “X” count is equal to the total ship length, that player’s whole ships are sunk. If other player still has swimming parts, this means s/he has won the game. If both of their ships are all sunk, the game is a draw. Function will return a string that indicates the situation.

Checking Ship States

Using ship_states() functions, the state of each ship for each player will be displayed. This function will take no parameter and return a string that shows the swimming/sunk status of each ship. By checking the dictionaries from global scope that holds the information of each ship for each player, a string with appropriate spacing will be returned to be used in print_table() function.

Checking if a Single Ship is Sunk

Carrier, Destroyer and Submarine exist as a single ship for each player. To check if these ships are sunk or not, check_single_ships() function will be used. It will take the grid that has the placements of the ships, grids that the attacks of the opponents are recorded, an integer that represents the current player, a string to represent the ship to be checked, and the length of that ship as arguments. It will look both of the grids for each player. The grids that contains the placements of the ships for a player will be iterated over and if any part of the given ship is encountered, it will check the other grid to see if that square took a hit. If it took a hit, a counter will be incremented by one. After the iteration if the counter value is equal to the length of that ship, the dictionary that hold the sinking/swimming information for the ships will be updated for the given ship by changing “-” to “X” in the values in that dictionary for name of the ship as a key.

Checking if a Multiple Ship is Sunk

Battleship and Patrol Boat exist at multiple numbers for both player. Their positions are hold in a list in the global scope. To check any of them is sunk or not, every one of the ship will be checked individually. By iterating over the list that contains the positions for these ships, each one of them will be checked. By looking at their position in the hidden grids, total hit will be calculated by the number of appearances of “X” in that ship position and if total hit is equal to the length of that ship, this will mean that, that ship is sunk. In this case, one of the “-” in the dictionary in the global scope will be replaced with “X” to show one of the ship is sunk.

Checking IOError While Running the Program

Everything for this program to work depends on the files that are given as a command line argument. If any of them does not work, the game will be unplayable. To check this, a function called io_error() will be used. It will iterate over each file in the command line and try to open them. If any of them cause an error, it will cause it an record that problematic file. At the and of the process, it will print a message about problematic files if they exist. Otherwise, program will continue with the next instructions.

Checking IndexErrror of a Given Shot

While playing the game with the play() function, that function will check if the current player’s current shot is available or not. It can miss row value such as “,D;” or column value such as “1,;” or both such as “,;” and so on. To check missing values, index_error() function will be used. It will check if the given shot is proper or not. If that shot is proper, it will return 0. Otherwise, it will return a string that explains why that shot is not suitable. For example, “1,;” will return “Missing row value! Given value: 1,”

Checking ValuError of a Given Shot

A shot could look like containing both row and column value, however, they could be wrong values. To check this a function called value_error() will be used. It will check if the given shot is proper or not. It may contain invalid row value or column value or both. Also, it can contain more than 2 values to be used as a row and column values. Such as, “1,1;”, “A,A;”, “1;A1;A;”. First two do not miss any value but they contain improper row and column values. Last one has too many values to be used as a row and column value. According to the given shots, function will return a informative string that says why they are not proper if they are improper, otherwise, it will return 0.

Checking if Overlap Occurs or a Given Input is Valid

To check if an overlap occurs, the situation that a single square containing more than one ship parts, or a given inputs that represent the ships are valid or not, check_overlap_and_correctness() function will be used. This function will iterate over the list that contains ship placements, and if it finds a square that has more than 1 character inside of it or if it finds a character other than “C, B, S, P, D” inside of the square, it will raise and exception and print a message to file and console by terminating the game.

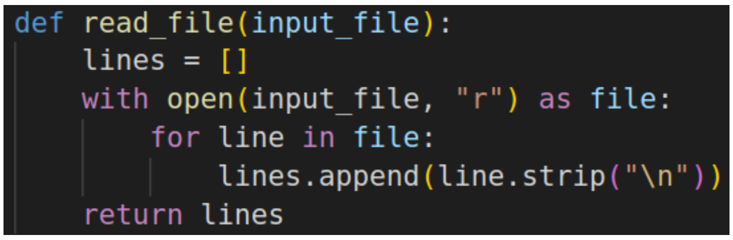
Writing Outputs to Console and File

After each operation a descriptive message was formed and appended to a list called file_messages that is in the global scope. To form a final single string of messages, each messages are combined using join() method. Using print() statement this final message is printed to console and using write_file() function, it will printed to an output file.

PROGRAMMER’S CATALOG

def read_file(input_file):

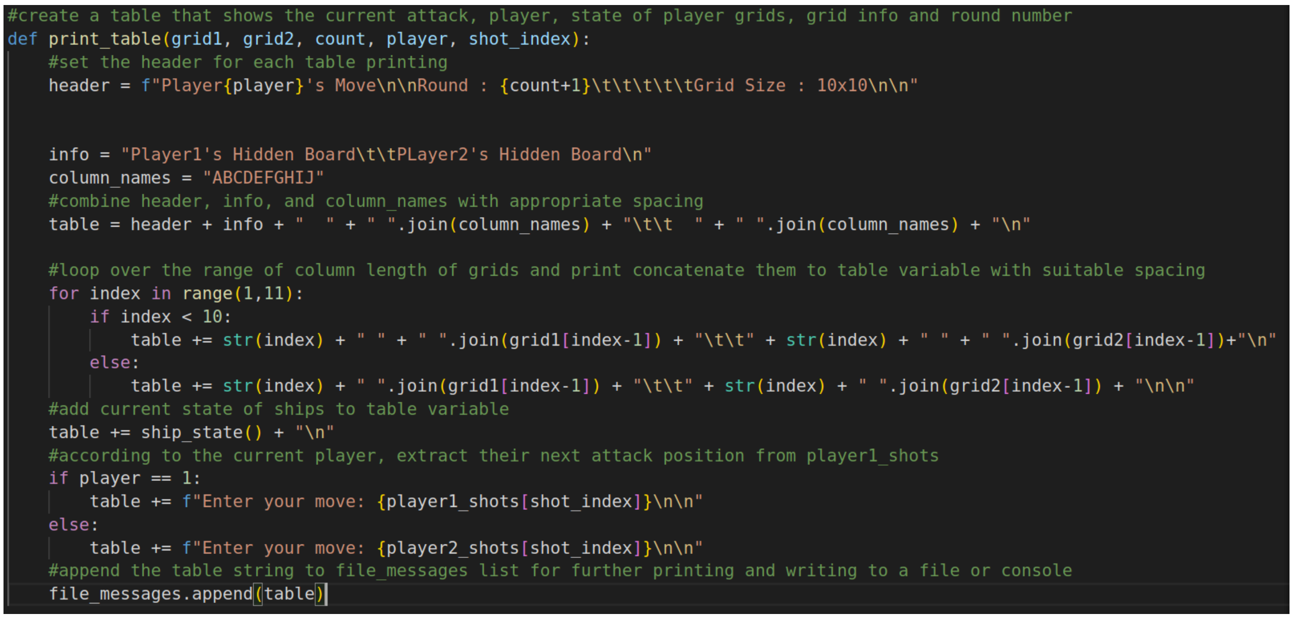
This function takes a file name as a parameter and reads it line by line. An empty list is created inside of the function to hold the information of each line from the file. Using with statement, file is opened in “r” read mode and each line is iterated over in the file with for loop. New line character is stripped from at the end of each line and each line is appended to a list. Finally, the list that holds the information of each line in the file is returned.



```python
def read_file(input_file):
    lines = []
    with open(input_file, "r") as file:
        for line in file:
            lines.append(line.strip("\n"))
    return lines
```

def print_table(grid1, grid2, count, player, shot_index):

This function will take 5 parameters. grid1 is a multi-dimensional Python list that holds the state of first player’s hidden grid after the shots from second player. grid2 is a multi-dimensional Python list that holds the state of second player’s hidden grid after the shots from first player. count is an integer that represents the current round number. player is an integer that represents the current player, 1 for first player, 2 for second player. shot_index is an integer that represents the index of the current players shot in his/her list of shots.

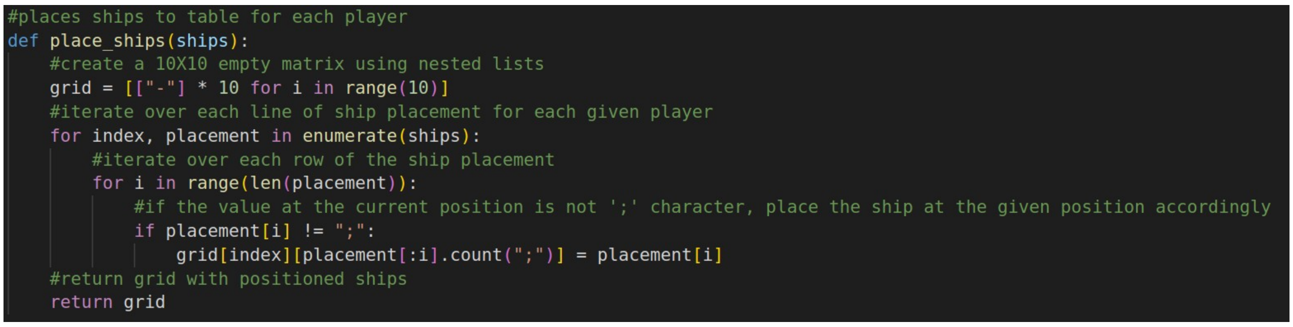


```python
#create a table that shows the current attack, player, state of player grids, grid info and round number
def print_table(grid1, grid2, count, player, shot_index):
    #set the header for each table printing
    header = f"Player{player}'s Move\n\nRound : {count+1}\t\t\t\t\tGrid Size : 10x10\n\n"


    info = "Player1's Hidden Board\t\tPLayer2's Hidden Board\n"
    column_names = "ABCDEFGHIJ"
    #combine header, info, and column_names with appropriate spacing
    table = header + info + "  " + " ".join(column_names) + "\t\t  " + " ".join(column_names) + "\n"

    #loop over the range of column length of grids and print concatenate them to table variable with suitable spacing
    for index in range(1,11):
        if index < 10:
            table += str(index) + " " + " ".join(grid1[index-1]) + "\t\t" + str(index) + " " + " ".join(grid2[index-1])+"\n"
        else:
            table += str(index) + " ".join(grid1[index-1]) + "\t\t" + str(index) + " ".join(grid2[index-1]) + "\n\n"
    #add current state of ships to table variable
    table += ship_state() + "\n"
    #according to the current player, extract their next attack position from player1_shots
    if player == 1:
        table += f"Enter your move: {player1_shots[shot_index]}\n\n"
    else:
        table += f"Enter your move: {player2_shots[shot_index]}\n\n"
    #append the table string to file_messages list for further printing and writing to a file or console
    file_messages.append(table)
```

Using given parameters and spacing, appropriate header, info and column_names variables are created to give a descriptive outcome when printing. By iterating from 1 to 10, both included, each row from both player’s grid are extracted and concatenated with the table variable that holds the descriptive message. Appropriate spacing is achieved using if/else statements and “\t” character. After that, to be able to see the current sunk/swimming information of each ship, ship_state() function is called and its result is concatenated to the table as well. To extract the shot of the given player, shot_index value used as a list index in player*_shots list(* represent player parameter), which contains the shots of the given player and resultant string is concatenated to table variable . Final string value is appended to a list called final_messages for further printing operations.

def place_ships(ships):

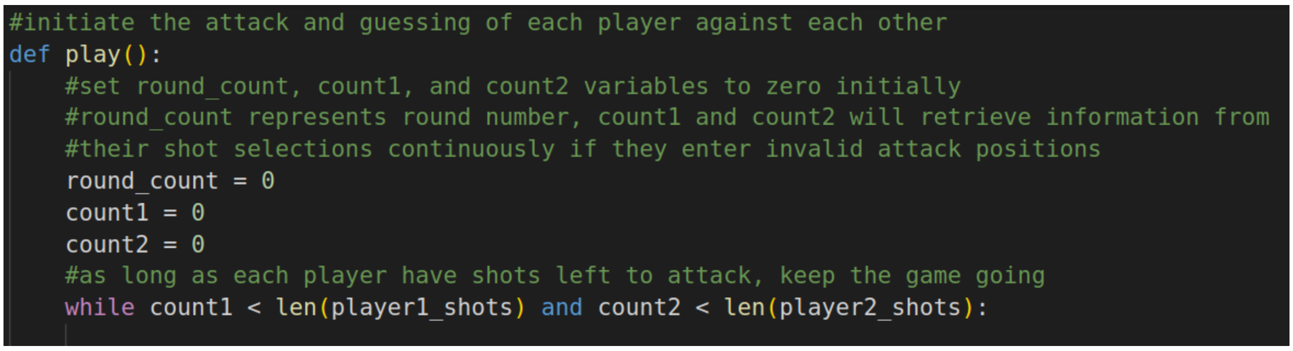
This function enables the game to be played. It takes a parameter called ships, which is a multi-dimensional Python list that contains the placements of ships for individual players. Ship placements for each player is given inside of a file as a command line argument. File names are obtained by sys.argv[i] (i represents an integer) and information inside the files are extracted with read_file() function. Resultant list from read_file() function contains each line as a string. Ship placement files consists of 10 lines and each line have 9 “;” characters. Between these characters a letter is placed to represent a piece of a ship. This function extracts that information and places those letters in each individual player’s grids. Firstly, an empty multi-dimensional Python list is created using list comprehension to be filled. Each list inside of the main list contains 10 “-” characters to represent empty spaces. Using enumerate and a for loop, given parameter list is iterated over. And using nested for loops, each string inside of the list is checked. If the current character at the current string is not a “;” character, this means a piece of ship is encountered. The number of “;” characters up until that index in the current string gives the column value of the piece of that ship. placement variable is a string that holds the information from each line in the ships placements files and index is an integer that gives the index of placement variable. Using the obtained column value and index variable, that piece of ship is placed at the obtained position in the created empty multi-dimensional Python list. This process is carried out until given ships list completely iterated over and each ship part is placed in the empty grid. After that, the grid with filled ship parts are returned for further use.



```python
#places ships to table for each player
def place_ships(ships):
    #create a 10X10 empty matrix using nested lists
    grid = [["-"] * 10 for i in range(10)]
    #iterate over each line of ship placement for each given player
    for index, placement in enumerate(ships):
        #iterate over each row of the ship placement
        for i in range(len(placement)):
            #if the value at the current position is not ';' character, place the ship at the given position accordingly
            if placement[i] != ";":
                grid[index][placement[:i].count(";")] = placement[i]
    #return grid with positioned ships
    return grid
```

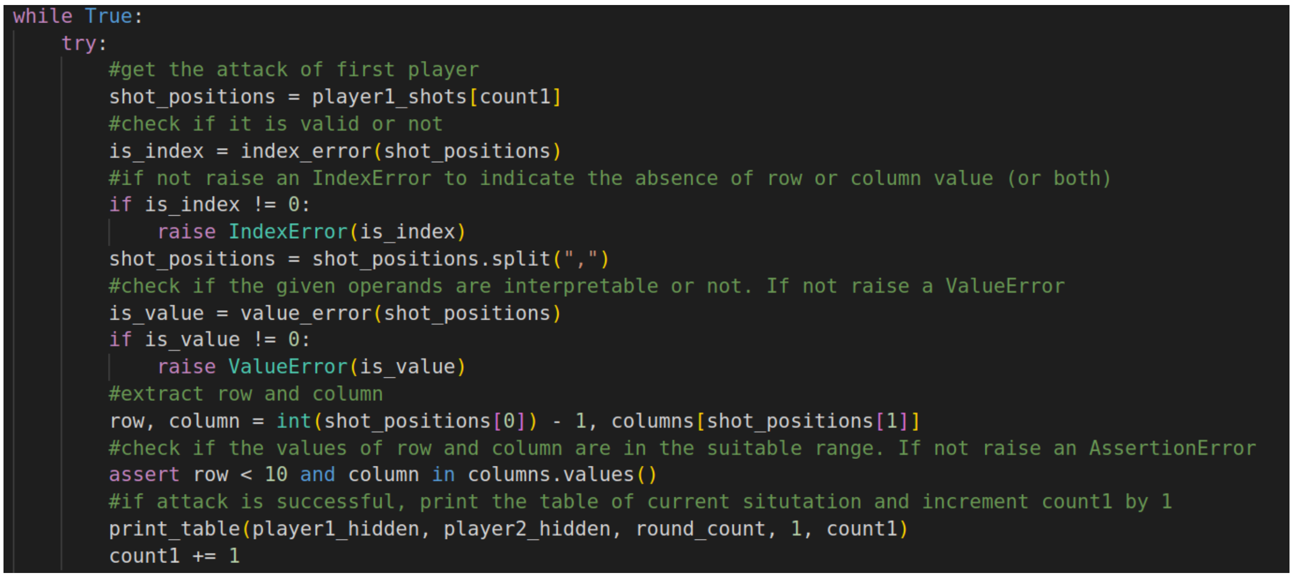
def play():

After obtaining the ship placements and attacks of each players for the whole game from the files given from the command line, the game is ready to be played. And what play() function does is basically that; simulating each round, getting the attacks of each player, updating the grids of each player, updating the state of sunk/swimming of each ship and announcing the winner of the game if there is one.



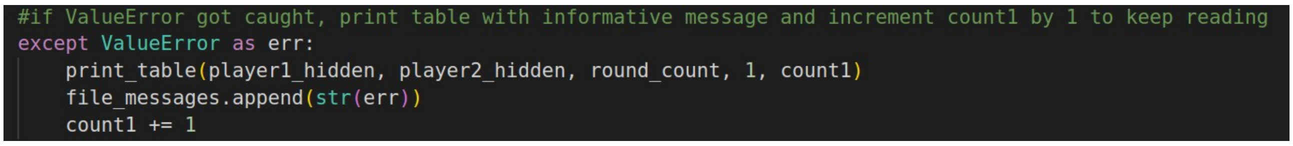
```
#initiate the attack and guessing of each player against each other
def play():
    #set round_count, count1, and count2 variables to zero initially
    #round_count represents round number, count1 and count2 will retrieve information from
    #their shot selections continuously if they enter invalid attack positions
    round_count = 0
    count1 = 0
    count2 = 0
    #as long as each player have shots left to attack, keep the game going
    while count1 < len(player1_shots) and count2 < len(player2_shots):
```

At the start of the function, round_count, count1, and count2 variables are created for the control flow of the function. round_count will hold the information of which round is currently being played right now. count1 and count2 will store the index of the shots that each player has taken in each round. count1 and count2 variables is created and used instead of simply using round_count variables to extract the attacks of each player from their respective shot list, because, if an invalid shot is encountered in one of the players list, reading from the same list should continue without changing the number of round number, namely, round_count. For this purpose, round1 and round2 variables have a huge responsibility. Initial while loop is created for simulating the game as long as both players have a shot to take in their lists. Otherwise, game will halt. To start the game, Player1 will be asked to take the first shot. Using count1 variable, his/her shot for the round is accessed from the list in the global scope that holds the whole shots for the game. After getting the shot, which is in the form of "row,column", validity of the shot is being checked using several functions. Firstly, to see if a given shot has missing row or column value or both, index_error() function is called. This function will return a string that explains what is the problem about the given shot is if a shot is problematic, and will return 0 if it is a suitable one. By checking the returned value, IndexError is raised depending on that value. Otherwise, other checking operations is performed.



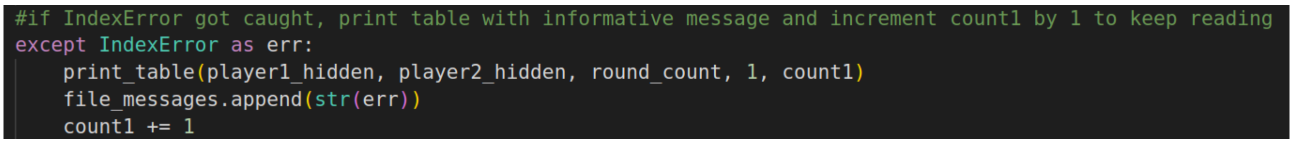
```
while True:
    try:
        #get the attack of first player
        shot_positions = player1_shots[count1]
        #check if it is valid or not
        is_index = index_error(shot_positions)
        #if not raise an IndexError to indicate the absence of row or column value (or both)
        if is_index != 0:
            raise IndexError(is_index)
        shot_positions = shot_positions.split(",")
        #check if the given operands are interpretable or not. If not raise a ValueError
        is_value = value_error(shot_positions)
        if is_value != 0:
            raise ValueError(is_value)
        #extract row and column
        row, column = int(shot_positions[0]) - 1, columns[shot_positions[1]]
        #check if the values of row and column are in the suitable range. If not raise an AssertionError
        assert row < 10 and column in columns.values()
        #if attack is successful, print the table of current situtation and increment count1 by 1
        print_table(player1_hidden, player2_hidden, round_count, 1, count1)
        count1 += 1
```

In the situation of no IndexError is raised, given shot is split by comma to extract the row and column value separately inside of a list. After that, is_value() function is called with these values to see if given row and column values are in the suitable range. is_value() function will check if the row value is in the range of 1 to 10 and column value is the range of A to J. If any of them fails or some weird value is given to the function that has been seen as proper by is_index() function, is_value() function will return a string explaining what is the issue about the given value. Otherwise, it will return 0 to indicate everything is good. ValuError will be raised depending on the returned value from is_value() function. After that, some adjusting will be performed to these values. When a player given 1,A as a shot, the index of the 1 given as a row will be 0 in the grids of each player. By subtracting 1 from this value, we will obtain proper index for the shot. And columns values are consisting of letters, however, we need integers to index a list. Using the columns dictionary in the global range, we obtain the correct value for that value as well. To check the final validity of the shot, we assert a statement for each shot; each row value need to be less than 10 (the row mentioned here is the value obtained by subtracting one from the player's input) and given column value has to be in the mentioned dictionary. If this assert statement fails, AssertionError will be raised and game will continue from there. Otherwise, the shot passes all the checking operations and ready to be used in the game. By calling print_table() function with the necessary functions, shot is fired at the opponent's table and proper operations executed accordingly mentioned in the description of the print_table() function. Finally, count1 variable is incremented by 1 for the next round.



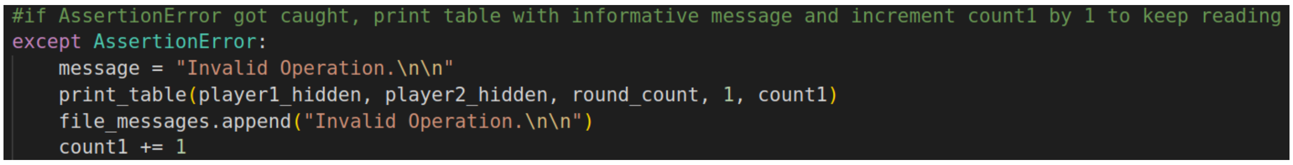
```
#if ValueError got caught, print table with informative message and increment count1 by 1 to keep reading
except ValueError as err:
    print_table(player1_hidden, player2_hidden, round_count, 1, count1)
    file_messages.append(str(err))
    count1 += 1
```

When a ValueError is caught, the error message is caught from the is_value() function and print_table() function is called to print the latest state of each grid. After that, this message is added to a list called file_messages for final printing and count1 variable is incremented by 1 to continue to read from the same player.



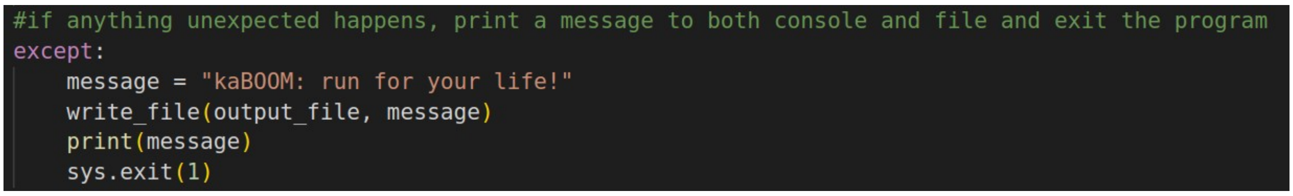
```
#if IndexError got caught, print table with informative message and increment count1 by 1 to keep reading
except IndexError as err:
    print_table(player1_hidden, player2_hidden, round_count, 1, count1)
    file_messages.append(str(err))
    count1 += 1
```

When an IndexError is caught, the same process for the ValueError is applied in here as well.



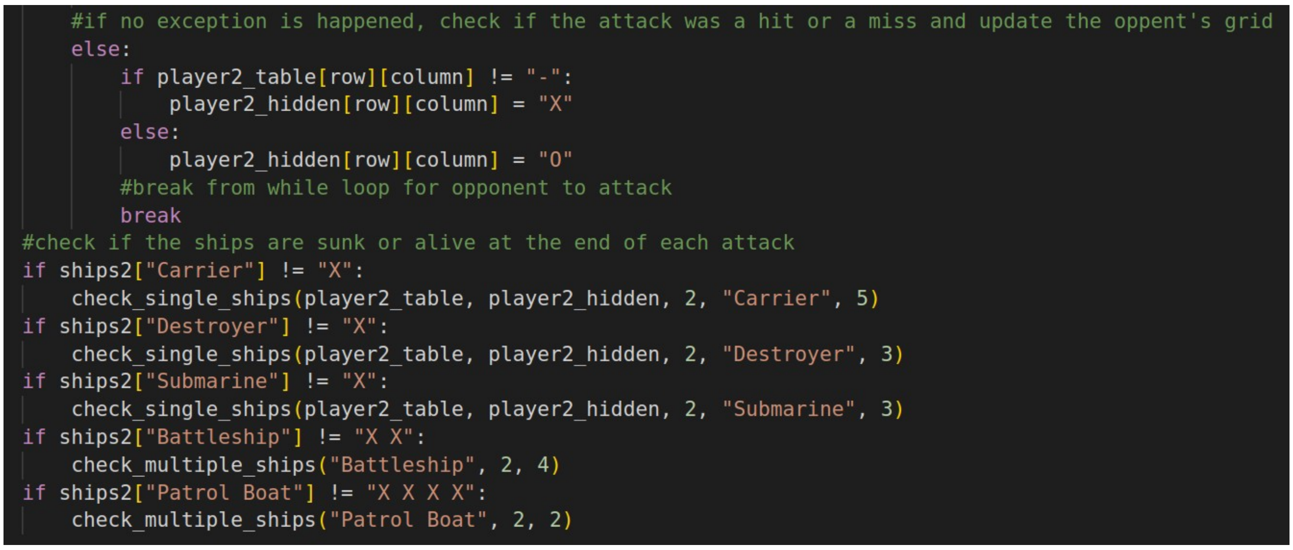
```
#if AssertionError got caught, print table with informative message and increment count1 by 1 to keep reading
except AssertionError:
    message = "Invalid Operation.\n\n"
    print_table(player1_hidden, player2_hidden, round_count, 1, count1)
    file_messages.append("Invalid Operation.\n\n")
    count1 += 1
```

In case of an AssertionError, logic is the same as the others, however, the error message is formed manually in this message. Other than that, count1 variable is incremented to continue reading from the same player.



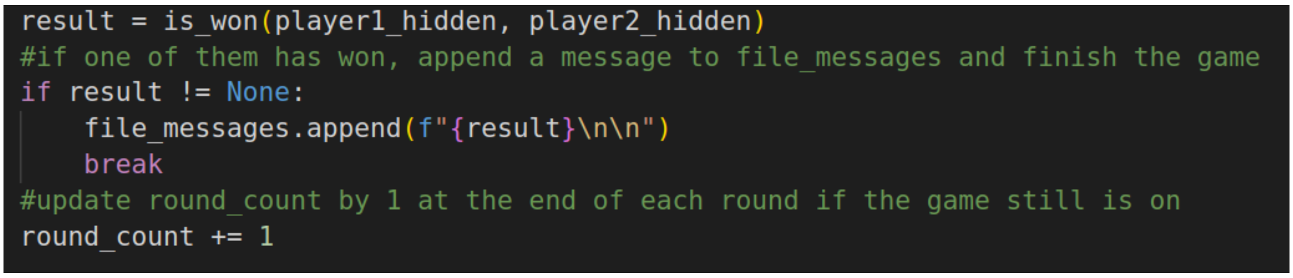
```
#if anything unexpected happens, print a message to both console and file and exit the program
except:
    message = "kaBOOM: run for your life!"
    write_file(output_file, message)
    print(message)
    sys.exit(1)
```

If anything unexpected happens, this except statement will catch it. In that case, an error message will be printed to both file and console and using sys.exit(1) statement program will be terminated.



```python
    #if no exception is happened, check if the attack was a hit or a miss and update the oppent's grid
    else:
        if player2_table[row][column] != "-":
            player2_hidden[row][column] = "X"
        else:
            player2_hidden[row][column] = "O"
        #break from while loop for opponent to attack
        break
#check if the ships are sunk or alive at the end of each attack
if ships2["Carrier"] != "X":
    check_single_ships(player2_table, player2_hidden, 2, "Carrier", 5)
if ships2["Destroyer"] != "X":
    check_single_ships(player2_table, player2_hidden, 2, "Destroyer", 3)
if ships2["Submarine"] != "X":
    check_single_ships(player2_table, player2_hidden, 2, "Submarine", 3)
if ships2["Battleship"] != "X X":
    check_multiple_ships("Battleship", 2, 4)
if ships2["Patrol Boat"] != "X X X X":
    check_multiple_ships("Patrol Boat", 2, 2)
```

If everything goes smoothly without causing any error, else block will be executed. Initial grid to hold the places of ships in secret from opponent is checked with the given row and column value. If there is a piece of a ship, the hidden grids to hold the attacks of opponents will be updating by changing “-” to “X” in that position. Otherwise, “-” will be changed to “O” to indicate the miss and using break statement initial infinite loop will be finished to continue with the other player. After each round from a player, opponent’s ships will be checked to see if they are swimming or any of them are sunk. By looking at the ships2 dictionary in the global scope, if all of the ships for given ship are not sunk, check_single_ships() function will be called to see if the final attack caused a sinking for any of the ships. Same process will be carried out for the second player almost similarly, just the name lists and dictionaries will vary, naturally they store values for each individual player respectively.

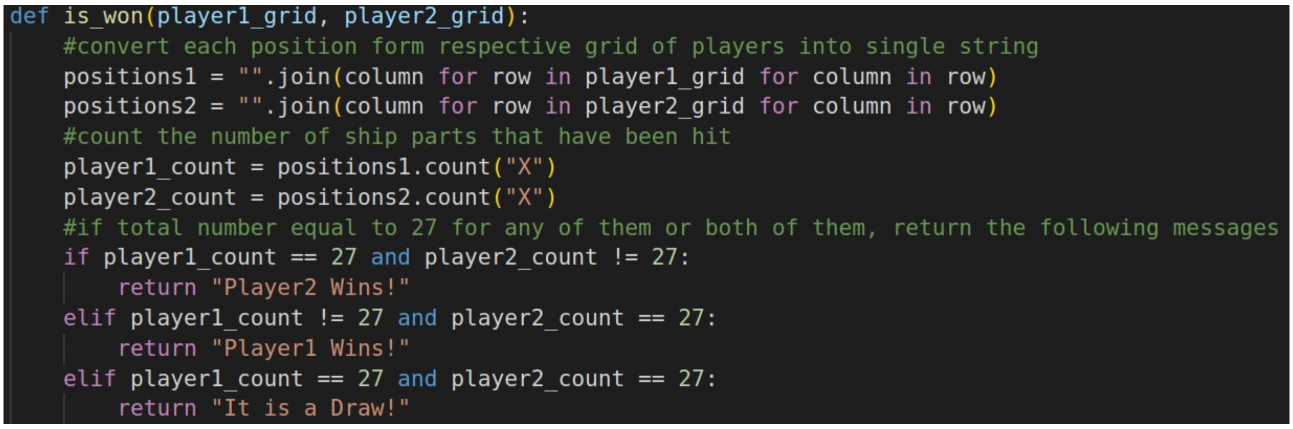


```python
result = is_won(player1_hidden, player2_hidden)
#if one of them has won, append a message to file_messages and finish the game
if result != None:
    file_messages.append(f"{result}\n\n")
    break
#update round_count by 1 at the end of each round if the game still is on
round_count += 1
```

After applying the process mentioned above, by calling is_won() function, winner is decided upon the returned value of that function. If returned value is not None, a winner is emerged. In that case, the returned string value that declares the winner is appended to the list for final printing operation. Using break statement, outermost while loop is broken and play() function is terminated. If there is no winner, round_count is incremented and game is on to be played with the same logic until there is a winner or a player runs out of attacks.

def is_won(player1_grid, player2_grid):

This function will take 2 parameters, both of them are a multi-dimensional Python list. This lists contains the state of each grid after each opponent took a shot at his/her opponent's grid. The number of appearances of "X" in each grid will give the total hit number. By default, each player has length of 27 as a total length of all ships combined. If a grid has 27 "X", that means, that player's all ships are sunk.

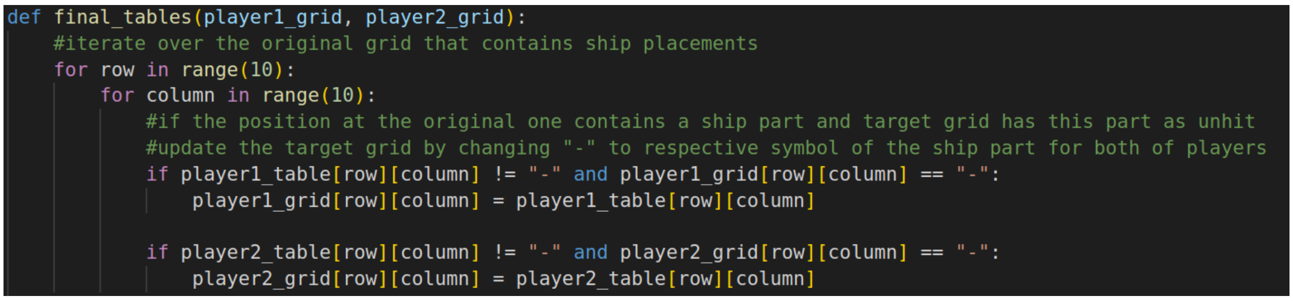


```python
def is_won(player1_grid, player2_grid):
    #convert each position form respective grid of players into single string
    positions1 = "".join(column for row in player1_grid for column in row)
    positions2 = "".join(column for row in player2_grid for column in row)
    #count the number of ship parts that have been hit
    player1_count = positions1.count("X")
    player2_count = positions2.count("X")
    #if total number equal to 27 for any of them or both of them, return the following messages
    if player1_count == 27 and player2_count != 27:
        return "Player2 Wins!"
    elif player1_count != 27 and player2_count == 27:
        return "Player1 Wins!"
    elif player1_count == 27 and player2_count == 27:
        return "It is a Draw!"
```

Using join, each entry inside of every list inside of the main list is joined together to a final string. The count of "X" is obtained using .count() method. If the result is equal to 27 for a player and not for the other, that player is lost the game. If both of them are equal to 27, there is a draw. Depending on the situation, a message will be returned.

def final_tables(player1_grid, player2_grid):

This function will receive two multi-dimensional Python list as a parameter. Both is the hidden list for each player that are used to record the opponent's shot at them. The printing logic is the same with the print_table() function, however, for the final result, the parts that no shot hit throughout the game will be displayed with the letter for that ship in that square. To achieve this, the list that holds the initial positioning of ships in the global scope, is iterated over. If a piece of ship is encountered and if there is "-" character in that same index in the given list as a parameter, that means that piece is never took a hit. "-" character is updated in the given list with the value in the list that has the ship positioning.

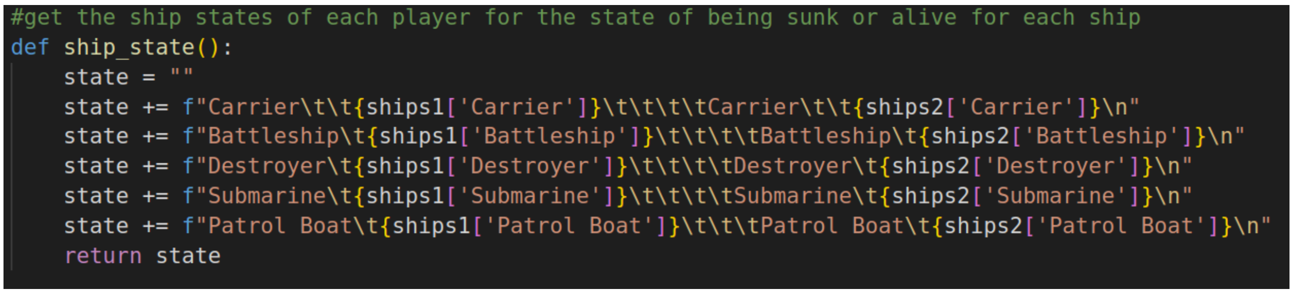


```python
def final_tables(player1_grid, player2_grid):
    #iterate over the original grid that contains ship placements
    for row in range(10):
        for column in range(10):
            #if the position at the original one contains a ship part and target grid has this part as unhit
            #update the target grid by changing "-" to respective symbol of the ship part for both of players
            if player1_table[row][column] != "-" and player1_grid[row][column] == "-":
                player1_grid[row][column] = player1_table[row][column]

            if player2_table[row][column] != "-" and player2_grid[row][column] == "-":
                player2_grid[row][column] = player2_table[row][column]
```

This is applied for both players, and after that necessary printing operations is carried out with suitable spacing and header information as it is applied in the print_table() function.

def ship_state():

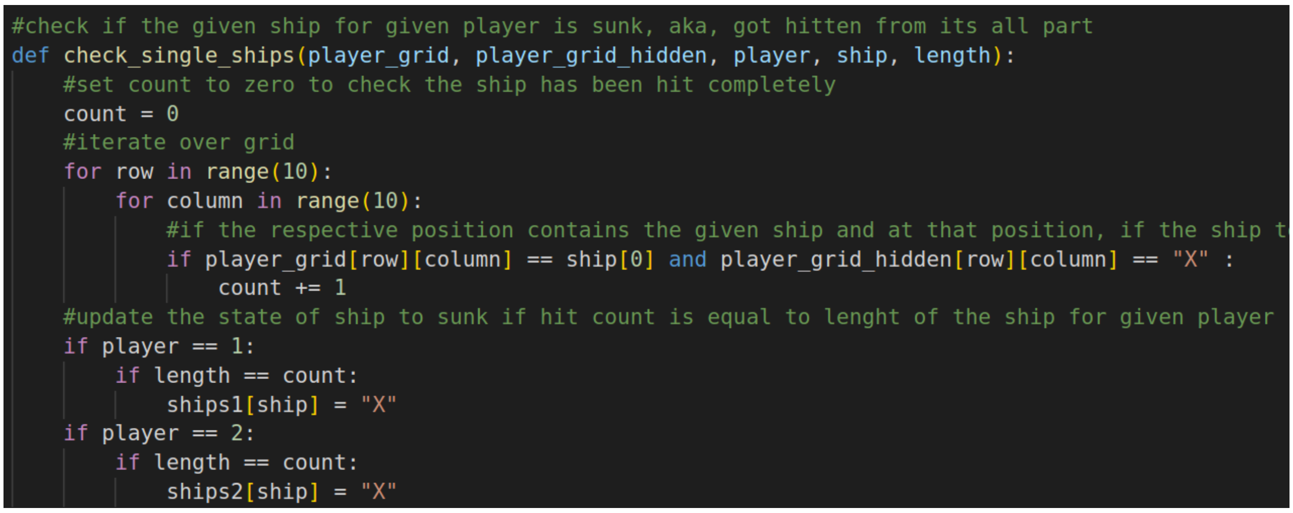
This function takes no parameter and return a string. Using the ships1 and ships2 dictionaries in the global scope, it will extracts the states of ships for each player from respective dictionaries and concatenates them to a string called state. As a result, this string in state variable is returned.



```python
#get the ship states of each player for the state of being sunk or alive for each ship
def ship_state():
    state = ""
    state += f"Carrier\t\t{ships1['Carrier']}\t\t\t\tCarrier\t\t{ships2['Carrier']}\n"
    state += f"Battleship\t{ships1['Battleship']}\t\t\t\tBattleship\t{ships2['Battleship']}\n"
    state += f"Destroyer\t{ships1['Destroyer']}\t\t\t\tDestroyer\t{ships2['Destroyer']}\n"
    state += f"Submarine\t{ships1['Submarine']}\t\t\t\tSubmarine\t{ships2['Submarine']}\n"
    state += f"Patrol Boat\t{ships1['Patrol Boat']}\t\t\tPatrol Boat\t{ships2['Patrol Boat']}\n"
    return state
```

def check_single_ships(player_grid, player_grid_hidden, player, ship, length):

In this function, the state of sinking/swimming will be checked for ships that are single ships for their type. Function will have 5 parameters. First two of them are the hidden grids and ships positions grid that are multi-dimensional Python list. player parameter is an integer either 1 or 2 that represent the player. ship is a string that gives the name of the ship. And length is an integer that holds the length of that given ship. Using this parameters, swimming/sunk state will be controlled.

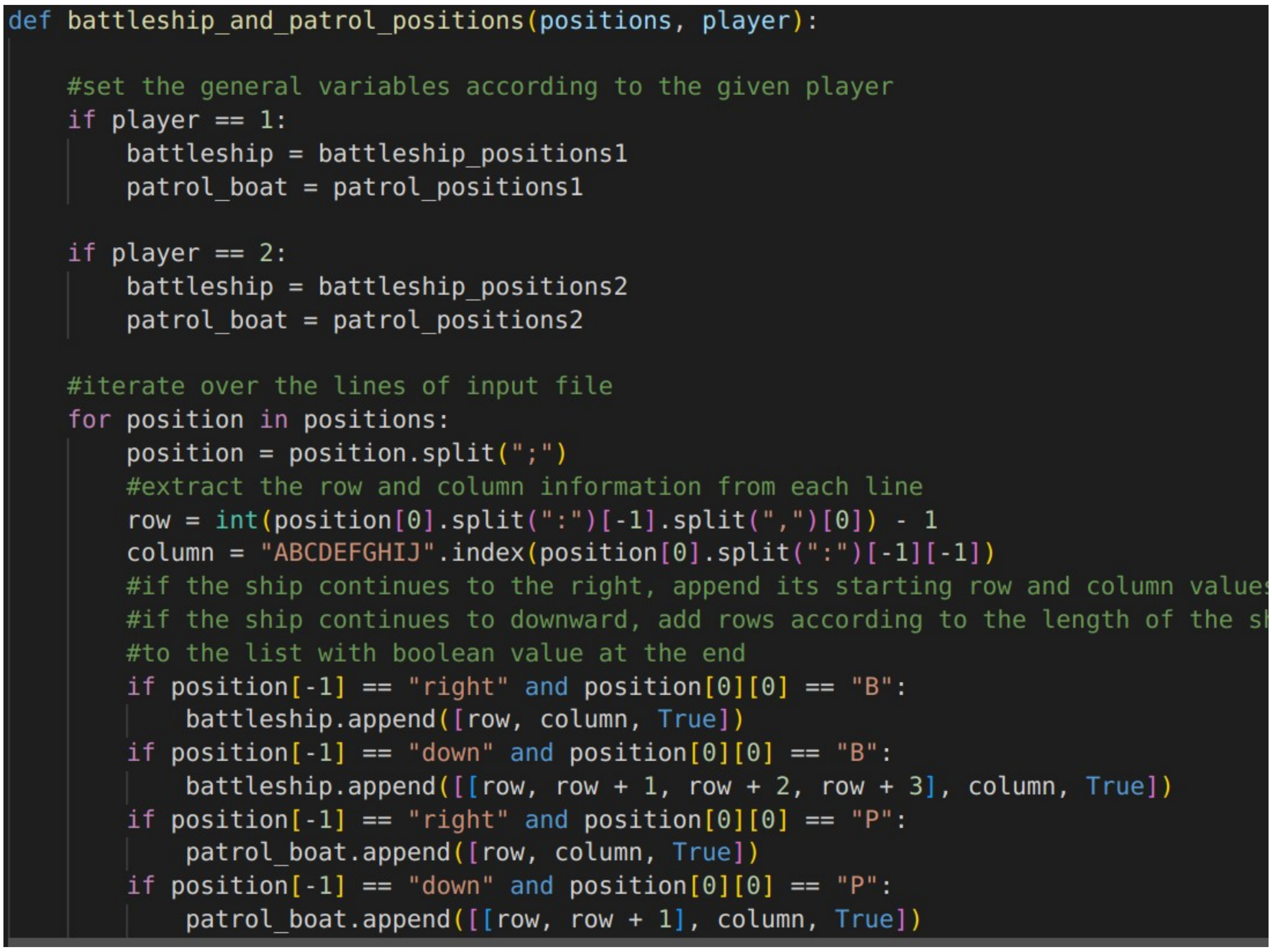


```python
#check if the given ship for given player is sunk, aka, got hitten from its all part
def check_single_ships(player_grid, player_grid_hidden, player, ship, length):
    #set count to zero to check the ship has been hit completely
    count = 0
    #iterate over grid
    for row in range(10):
        for column in range(10):
            #if the respective position contains the given ship and at that position, if the ship t
            if player_grid[row][column] == ship[0] and player_grid_hidden[row][column] == "X" :
                count += 1
    #update the state of ship to sunk if hit count is equal to lenght of the ship for given player
    if player == 1:
        if length == count:
            ships1[ship] = "X"
    if player == 2:
        if length == count:
            ships2[ship] = "X"
```

Firstly, initial positioning list will be iterated over for each square. If in a given square, there is a part of a ship that has given as a parameter, the hidden grid will be checked to see if that part took a hit or not. If so, count variable will be incremented, which holds the number of hit of that ship. If count is equal to the length of that ship, that means that ship is sunk. Therefore, state of that ship is updated in the dictionary in the global scope to “X” from “-” depending on the given player.

def battleship_and_patrol_positions(positions, player):

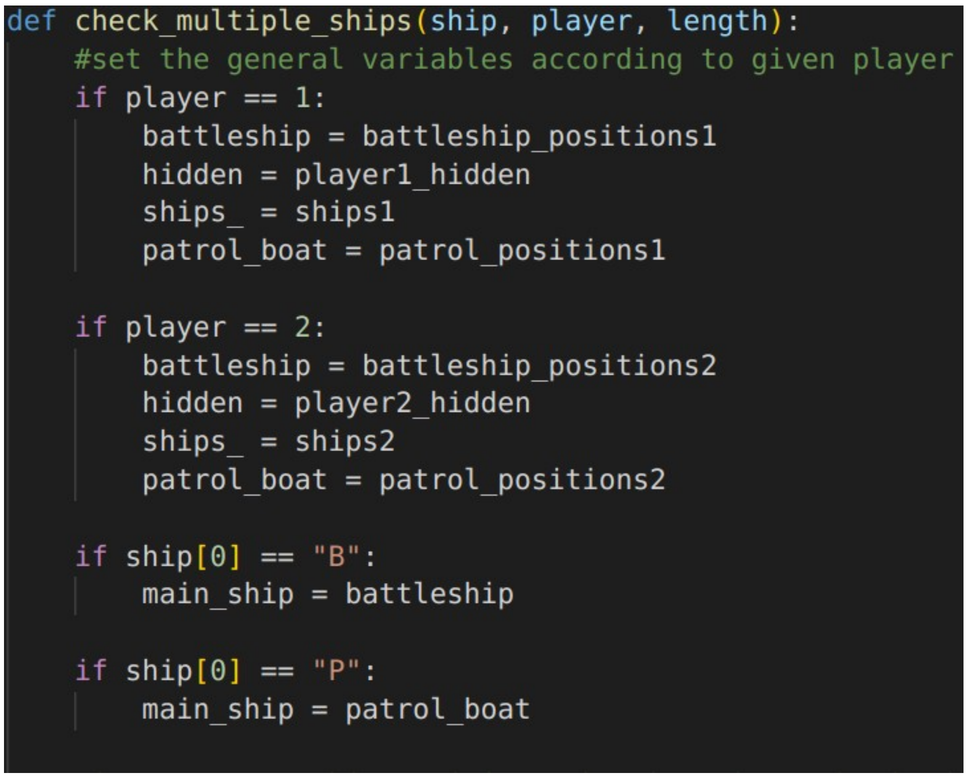
This function takes 2 parameters. First one is a list that contains strings, which are lines from Optional.txt files for each player. player is an integer that indicates the player. Depending on the player value, battleship and patrol_boat variables are assigned empty list in the global range. Iterating over the given positions parameter, positions of each ship for given player is investigated. Row and column information are extracted from each line by several splitting operations. Depending on the given alignment way of the ship and the type of ship, placement of the ship is appended to the suitable lists. If the alignment is “right”, starting row index, column index, and boolean True value is added to a list and this list is appended to the suitable main list mentioned above. If alignment is “down”, starting from the first row index, every row that particular ship has a piece is added to a list. Then, this list is added to another list with column value and boolean True value. Boolean value will be useful when checking this ships sinking status. Finally, this final list is appended to the suitable main list accordingly.



```
def battleship_and_patrol_positions(positions, player):

    #set the general variables according to the given player
    if player == 1:
        battleship = battleship_positions1
        patrol_boat = patrol_positions1

    if player == 2:
        battleship = battleship_positions2
        patrol_boat = patrol_positions2

    #iterate over the lines of input file
    for position in positions:
        position = position.split(";")
        #extract the row and column information from each line
        row = int(position[0].split(":")[-1].split(",")[0]) - 1
        column = "ABCDEFGHIJ".index(position[0].split(":")[-1][-1])
        #if the ship continues to the right, append its starting row and column value
        #if the ship continues to downward, add rows according to the length of the s
        #to the list with boolean value at the end
        if position[-1] == "right" and position[0][0] == "B":
            battleship.append([row, column, True])
        if position[-1] == "down" and position[0][0] == "B":
            battleship.append([[row, row + 1, row + 2, row + 3], column, True])
        if position[-1] == "right" and position[0][0] == "P":
            patrol_boat.append([row, column, True])
        if position[-1] == "down" and position[0][0] == "P":
            patrol_boat.append([[row, row + 1], column, True])
```

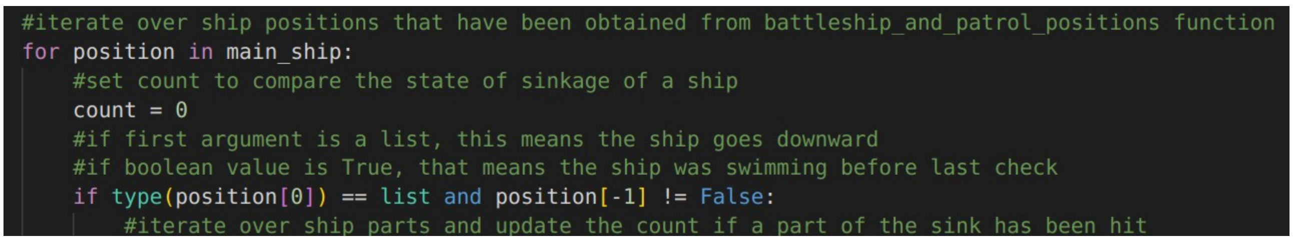
def check_multiple_ships(ship, player, length):

This function takes 3 parameters. First one is the name of the ship, a string. Second one is an integer which indicates which player's ship the given parameter is. Third one is an integer which gives the length of the ship. Depending on the given parameters, variables like battleship, ships_, main_ship,… etc. is created to simplify the code and get access to global data structures.



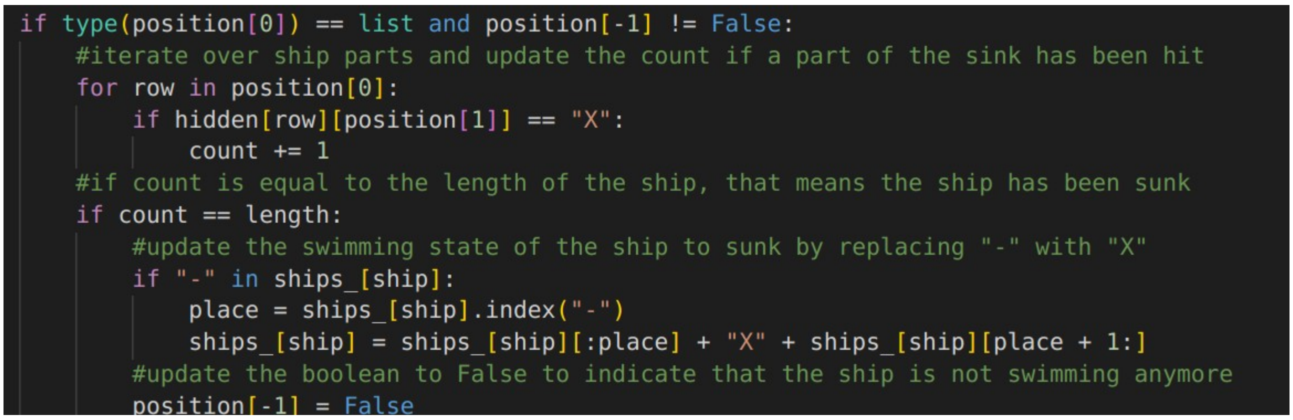
```python
def check_multiple_ships(ship, player, length):
    #set the general variables according to given player
    if player == 1:
        battleship = battleship_positions1
        hidden = player1_hidden
        ships_ = ships1
        patrol_boat = patrol_positions1

    if player == 2:
        battleship = battleship_positions2
        hidden = player2_hidden
        ships_ = ships2
        patrol_boat = patrol_positions2

    if ship[0] == "B":
        main_ship = battleship

    if ship[0] == "P":
        main_ship = patrol_boat
```

After setting up the variables, depending on the type of ship, main_ship variable is assigned to a list that contains the positions of that ship in the global scope. To check the status of each ship in that category, main_ship variable is iterated over.



```python
#iterate over ship positions that have been obtained from battleship_and_patrol_positions function
for position in main_ship:
    #set count to compare the state of sinkage of a ship
    count = 0
    #if first argument is a list, this means the ship goes downward
    #if boolean value is True, that means the ship was swimming before last check
    if type(position[0]) == list and position[-1] != False:
        #iterate over ship parts and update the count if a part of the sink has been hit
```

Each position in that list is checked. If the first item in the given position list has a type of list, this means that ship is aligned vertically and if its boolean value is True, that ship was swimming after last checking. After iterating over each part of the ship using a for loop and using row values from the first item in position list with column value from the second item from position list, if a part of that ship ship in the given position is hit by checking the hidden grid, count variable is incremented by 1. If at the end of the iteration, the count value is equal to the length, that means that ship is sunk. In that case, first "-" character in the ships_ dictionary that holds the state of ships is changed to "X" to showcase the sinking of that ship and True value is changed to False to show the sinking and avoiding further unnecessary checking when this function is called.

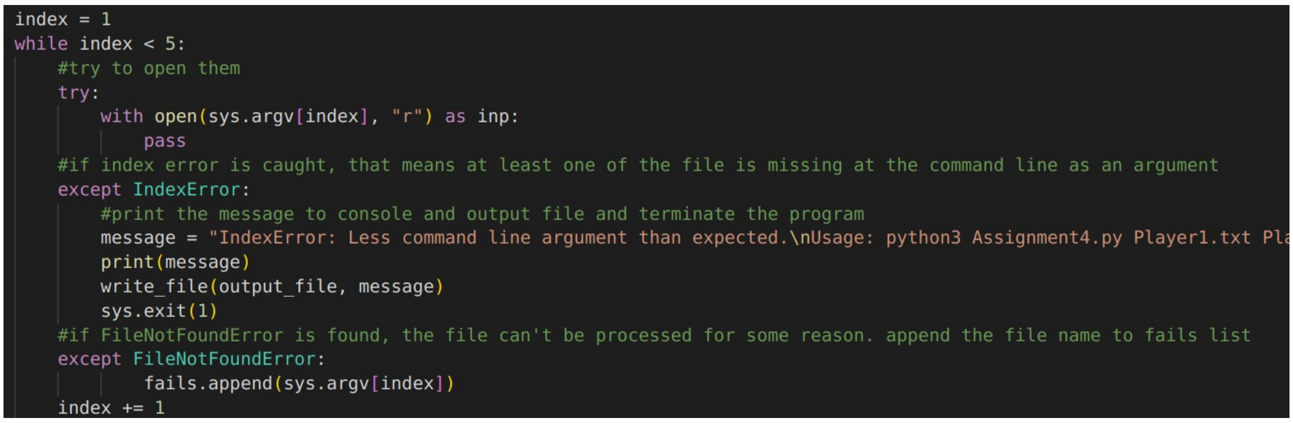


```
if type(position[0]) == list and position[-1] != False:
    #iterate over ship parts and update the count if a part of the sink has been hit
    for row in position[0]:
        if hidden[row][position[1]] == "X":
            count += 1
    #if count is equal to the length of the ship, that means the ship has been sunk
    if count == length:
        #update the swimming state of the ship to sunk by replacing "-" with "X"
        if "-" in ships_[ship]:
            place = ships_[ship].index("-")
            ships_[ship] = ships_[ship][:place] + "X" + ships_[ship][place + 1:]
        #update the boolean to False to indicate that the ship is not swimming anymore
        position[-1] = False
```

The same operation is applied for ships that have an alignment to the right, as the one above. The only differences here are the for loop and the type of the first item in the position list. Because these ships have right alignment, each part will have the same row value and incrementing column value, whereas, each part has the same column value and incrementing row value with the down aligned ships. Type of first item is an integer in this case. After applying the logic described above, if the ship is sunk, first available “-” character is changed to “X” in the ships_ dictionary in the global scope and the boolean value is updated to False for further easiness.

def io_error():

This function is the first function that has been called when the program is run at the beginning. It takes no parameter. It checks the availability of the given files from command line. Using a while loop and try/except blocks, it tries to open the command line files. If any file in the given index is missing, it raises an IndexError and prints a message to both file and console while terminating the program because with even one file missing, program will crash.

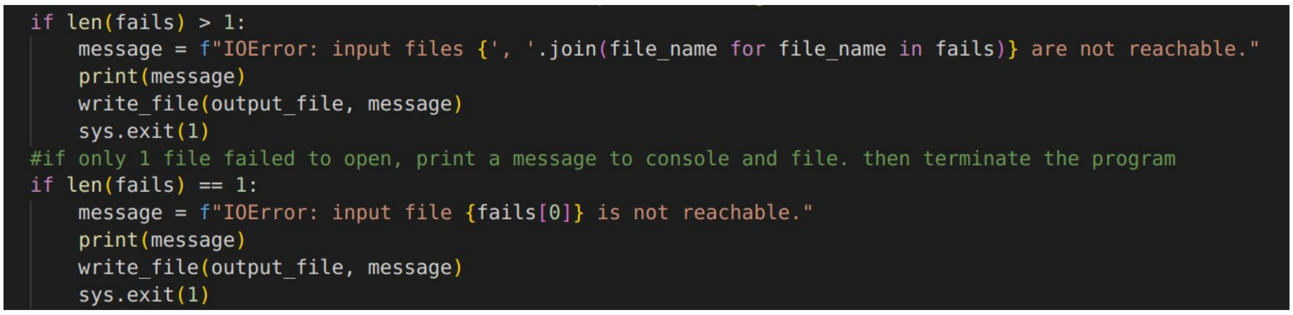


```
index = 1
while index < 5:
    #try to open them
    try:
        with open(sys.argv[index], "r") as inp:
            pass
    #if index error is caught, that means at least one of the file is missing at the command line as an argument
    except IndexError:
        #print the message to console and output file and terminate the program
        message = "IndexError: Less command line argument than expected.\nUsage: python3 Assignment4.py Player1.txt Pla
        print(message)
        write_file(output_file, message)
        sys.exit(1)
    #if FileNotFoundError is found, the file can't be processed for some reason. append the file name to fails list
    except FileNotFoundError:
            fails.append(sys.argv[index])
    index += 1
```

If every file is given, it will raise an FileNotFoundError if any of the files cannot be opened or reached for some reason. In that case, name of the file is appended to a list called fails to hold the names of each crashed file further printing.

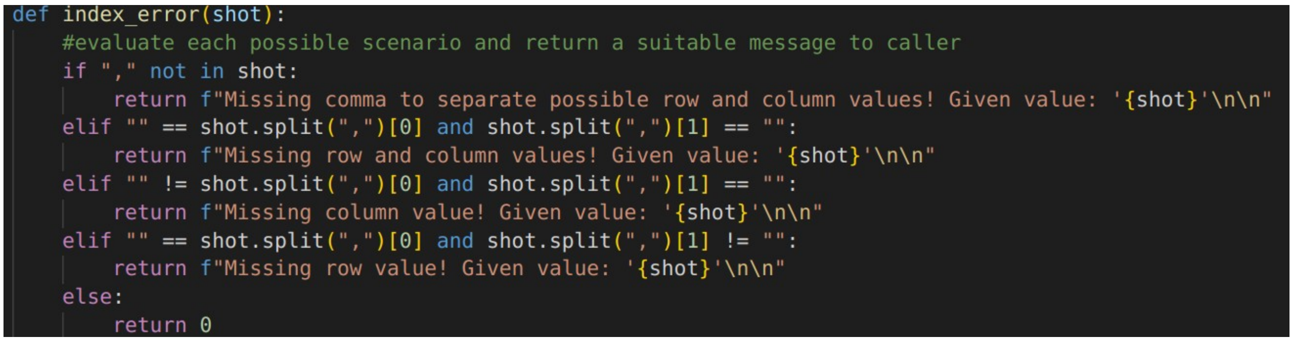
After obtaining each file possible crashed file, fails list is checked. If there are multiple files exist in the list, function will print a message to output file and to console containing the name of each file joined together.

If there is a single file inside of the fails list, function will print a message to console and output file describing that, that single file is failed to be opened for some reason. After this, program will be terminated using sys.exit(1) because without even a single file, program will not work. If fails list is an empty list, program will continue to function as intended.



```python
if len(fails) > 1:
    message = f"IOError: input files {', '.join(file_name for file_name in fails)} are not reachable."
    print(message)
    write_file(output_file, message)
    sys.exit(1)
#if only 1 file failed to open, print a message to console and file. then terminate the program
if len(fails) == 1:
    message = f"IOError: input file {fails[0]} is not reachable."
    print(message)
    write_file(output_file, message)
    sys.exit(1)
```

def index_error(shot):

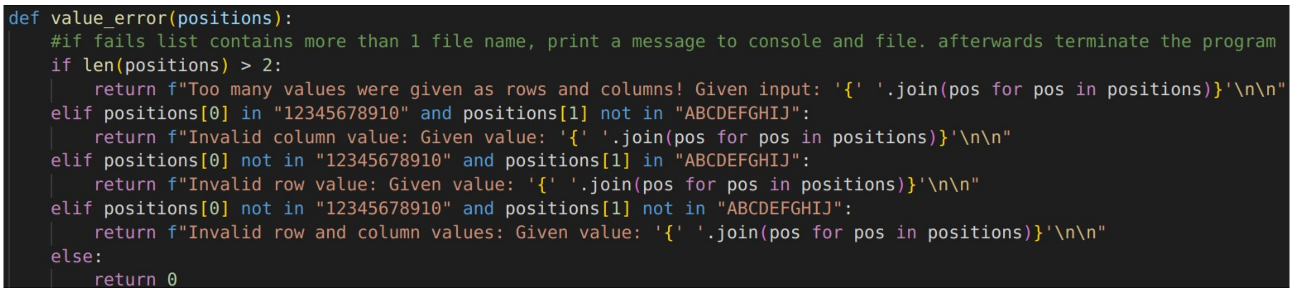


```python
def index_error(shot):
    #evaluate each possible scenario and return a suitable message to caller
    if "," not in shot:
        return f"Missing comma to separate possible row and column values! Given value: '{shot}'\n\n"
    elif "" == shot.split(",")[0] and shot.split(",")[1] == "":
        return f"Missing row and column values! Given value: '{shot}'\n\n"
    elif "" != shot.split(",")[0] and shot.split(",")[1] == "":
        return f"Missing column value! Given value: '{shot}'\n\n"
    elif "" == shot.split(",")[0] and shot.split(",")[1] != "":
        return f"Missing row value! Given value: '{shot}'\n\n"
    else:
        return 0
```

This function will be called from play() function. It will take 1 parameter that is the given shot for the current player at that round. Using if/elif/else statements, given string value will be checked to see that given shot has a missing row or column value or not. Depending on the shot, either a string with descriptive information or an integer, namely 0, will be returned.

def value_error(positions):

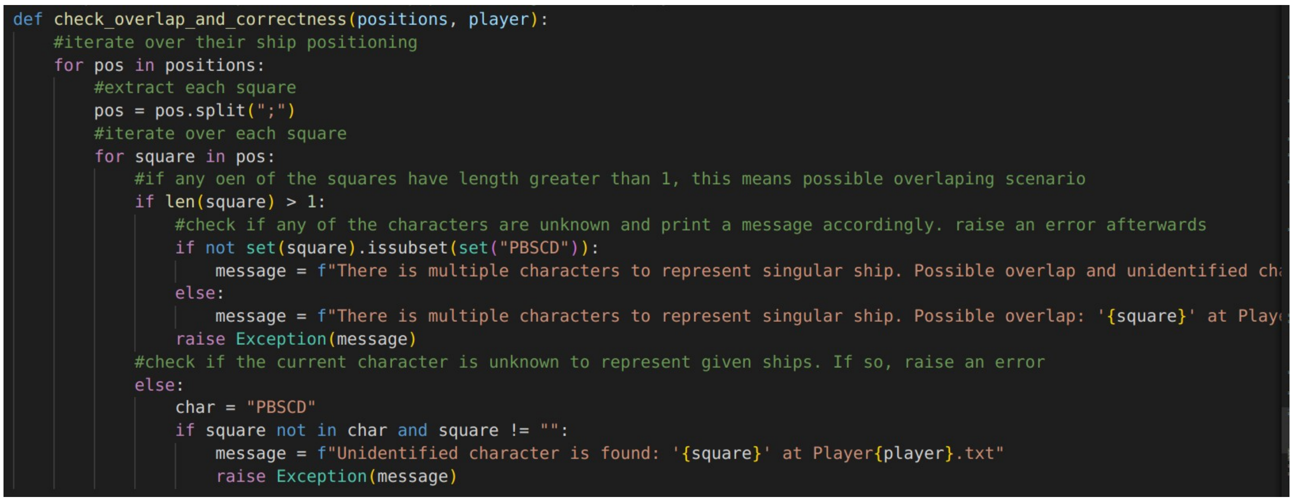
This function will take 1 parameter, namely, a list that holds the information of player's shot in that round. It will check the length of the list, row value suitability, and column suitability for the given shot. If list has more than 2 values, or its first item, namely row value, is not in the given range, or its second item, namely column value, is not in the given range, it will return a descriptive string message. Otherwise, it will return 0 to indicate it is a valid shot.



```python
def value_error(positions):
    #if fails list contains more than 1 file name, print a message to console and file. afterwards terminate the program
    if len(positions) > 2:
        return f"Too many values were given as rows and columns! Given input: '{' '.join(pos for pos in positions)}'\n\n"
    elif positions[0] in "12345678910" and positions[1] not in "ABCDEFGHIJ":
        return f"Invalid column value: Given value: '{' '.join(pos for pos in positions)}'\n\n"
    elif positions[0] not in "12345678910" and positions[1] in "ABCDEFGHIJ":
        return f"Invalid row value: Given value: '{' '.join(pos for pos in positions)}'\n\n"
    elif positions[0] not in "12345678910" and positions[1] not in "ABCDEFGHIJ":
        return f"Invalid row and column values: Given value: '{' '.join(pos for pos in positions)}'\n\n"
    else:
        return 0
```

def check_overlap_and_correctness(positions, player):

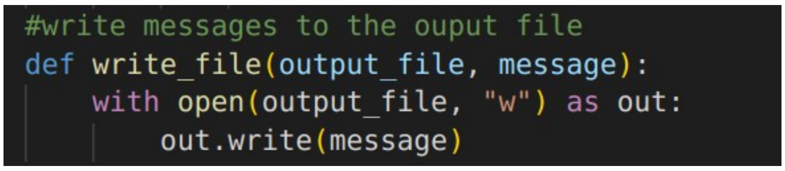
This function will take 2 parameters: player is an integer that indicates the given integer, 1 for Player1 and 2 for Player2; positions is a list that holds lines of ship placement file for the given player as a string for each line. Iterating over positions list, each string is split to extract the every value for each square in the grid. If extracted value has more than 1 character, and there is an unidentified character to represent a ship in that square, an exception will be raised with a descriptive message. To check this subset property of sets will be used. For example, let a set consists of letters "PBSCD" and square in the given position is a set with letters "CD". Square will be a subset of the given set, however, another set such as "CG" will not be subset. If length is still more than 1 and the characters represent ships in that square, an overlap occurred in that square. In this situation, an exception will be raised with a descriptive message. If there is 1 character but it is not an allowed one, an exception will be raised with a descriptive message related with the failure.



```
def check_overlap_and_correctness(positions, player):
    #iterate over their ship positioning
    for pos in positions:
        #extract each square
        pos = pos.split(";")
        #iterate over each square
        for square in pos:
            #if any oen of the squares have length greater than 1, this means possible overlaping scenario
            if len(square) > 1:
                #check if any of the characters are unknown and print a message accordingly. raise an error afterwards
                if not set(square).issubset(set("PBSCD")):
                    message = f"There is multiple characters to represent singular ship. Possible overlap and unidentified cha
                else:
                    message = f"There is multiple characters to represent singular ship. Possible overlap: '{square}' at Play
                raise Exception(message)
            #check if the current character is unknown to represent given ships. If so, raise an error
            else:
                char = "PBSCD"
                if square not in char and square != "":
                    message = f"Unidentified character is found: '{square}' at Player{player}.txt"
                    raise Exception(message)
```

def write_file(output_file, message):

This function enables users to understand and observe what was going on throughout the game. It will take 2 parameters: output_file, a string that holds the name of the output file and a message, a string that holds the whole process of the game. Using with expression, this function open the output file in write mode and writes the message to file.



```
#write messages to the ouput file
def write_file(output_file, message):
    with open(output_file, "w") as out:
        out.write(message)
```

GLOBAL DATA STRUCTURES AND OPERATIONS

In the global scope, before delving into playing game, some initializations of data structures and checking should be done. As a start, fails list is created to hold the names of the failed file names from the command line. Name of the output file is stored inside of a output_file variable and as mentioned before, io_error() function is called to see if the files are able to be opened or not. If files from command line passes this, further testing will be carried out.

```python
#create a list called fails to hold the file names of possible failed file's from trying to open them
fails = []
#set the output name of the file
output_file = "Battleship.out"
#check firstly the correctness of the files. If no problem occurs continue, else terminate the program
io_error()
```

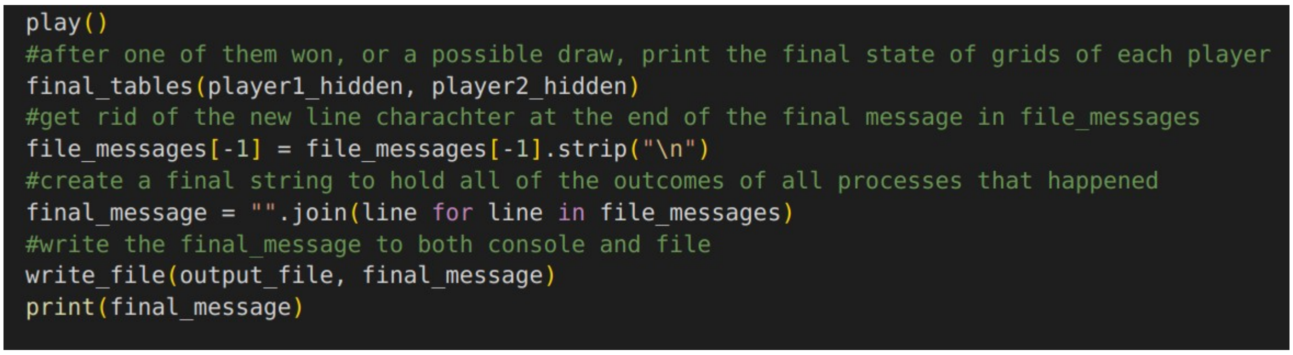
```python
try:
    check_overlap_and_correctness(read_file(sys.argv[1]), 1)
    check_overlap_and_correctness(read_file(sys.argv[2]), 2)
#if overlap occurs in one of the files, print a message to console and file. Then quit the program
except Exception as err:
    print(f"kaBOOM: run for your life! {err}")
    write_file(output_file, f"kaBOOM: run for your life! {err}")
    sys.exit(1)
```

Using try/except/else blocks check_overlap_and_correctness() function is called with the mentioned parameters. If that function throws an error, except block will catch it and output a message to console and output file before terminating the program.

If no error has been caught, else block will be executed. Inside of that block, players ships will be placed to grids, hidden empty grids will be formed to record the shots that opponent fired at to his/her ships. Shots of players will be stored inside of a list using described functions before. Welcoming message will be stored inside of a message variable. file_messages list will be initialized to hold the outputs of all processes. Battleship and Patrol Boat's positions will be stored in lists created for each player. columns dictionary will be created with letters as keys and integers from 0 to 9, both included, as values. ships1 and ships2 dictionaries will be created with ship names as keys and sinking indicators as values.

```python
else:
    #obtain ship placements and shots selection of each player
    player1_table = place_ships(read_file(sys.argv[1]))
    player2_table = place_ships(read_file(sys.argv[2]))
    player1_shots = read_file(sys.argv[3])[0].split(";")[:-1]
    player2_shots = read_file(sys.argv[4])[0].split(";")[:-1]


    #create empty grids to hold the information of each attack
    player1_hidden = [["-"] * 10 for i in range(10)]
    player2_hidden = [["-"] * 10 for i in range(10)]
    #set the welcoming message of the game
    message = "Battle of Ships Game\n\n"
    #initialize the list to hold the all outcomes of each operation
    file_messages = [message]
    #get the ship positions of battleship and patrol boats
    player1_optional = [position[:-1] for position in read_file("OptionalPlayer1.txt")]
    player2_optional = [position[:-1] for position in read_file("OptionalPlayer2.txt")]

    #initialize empty list to hold the row and column informaiton of battleships and patrol boats
    battleship_positions1 = []
    battleship_positions2 = []

    patrol_positions1 = []
    patrol_positions2 = []
```

After this whole initialization process, play() function will be called to kick-start the game. After play() function is terminated, final_tables() function will be called to print the final state of each player's grid. Each string inside of the file_messages list will be joined together by getting rid of the final new line character. Using write_file() function, this obtained final string will be written to the output file and using print statement, it is printed to the console.

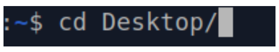


```
play()
#after one of them won, or a possible draw, print the final state of grids of each player
final_tables(player1_hidden, player2_hidden)
#get rid of the new line charachter at the end of the final message in file_messages
file_messages[-1] = file_messages[-1].strip("\n")
#create a final string to hold all of the outcomes of all processes that happened
final_message = "".join(line for line in file_messages)
#write the final_message to both console and file
write_file(output_file, final_message)
print(final_message)
```

USED DATA STRUCTURES

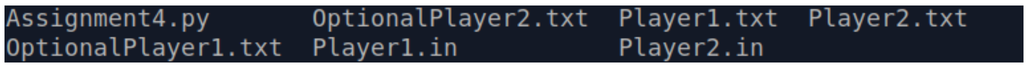
- player1_table and player2_table : Multi-dimensional Python list. Consists of 10 list and each list has 10 string values to hold the placements of ships
- player1_shots and player2_shots: Python list consisting of string values, which are the shots of each player for each round against their opponent.
- player1_hidden and player2_hidden: Multi-dimensional Python list. Consists of 10 list and each list has 10 string values. All strings inside of the lists are “-”. Initialized this way to record the shots of the opponent.
- battleship_positions1 and battleship_positions2: Multi-dimensional Python list that holds the position of the battleships for given player.
- patrol_positions1 and patrol_positions2: Multi-dimensional Python list that holds the position of the patrol boat for given player.
- columns: A dictionary consisting of letters as keys and integers as values. Keys range from A to J and values range from 0 to 9, both of the ranges inclusive.
- ships1 and ships2: Dictionaries that consist of ship names as keys (string) and state of ships as values(string). For example ships1[“Carrier”] = “-”.
- file_messages = A list consisting of strings, that holds the output of each operation throughout the game.

USER CATALOG

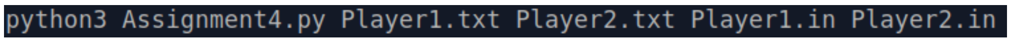


```
:~$ cd Desktop/
```

Before executing the program:

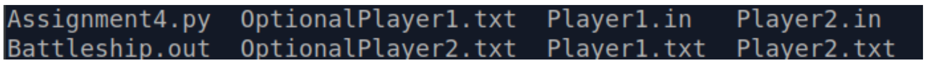


```
Assignment4.py       OptionalPlayer2.txt  Player1.txt  Player2.txt
OptionalPlayer1.txt  Player1.in           Player2.in
```



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

After executing the program:



```
Assignment4.py  OptionalPlayer1.txt  Player1.in   Player2.in
Battleship.out  OptionalPlayer2.txt  Player1.txt  Player2.txt
```

- Open the Terminal on your computer.
- Using cd command change your current directory to directory where your Assignment4.py exists.
- Be sure that Player1.txt, Player2.txt, Player1.in, Player2.in, OptionalPlayer1.txt, and OptionalPlayer2.txt files are in the same file location with your .py file.
- Otherwise, enter absolute path of those file as a command line argument when the program is run.
- Write python3 Assignment4.py Player1.txt Player2.txt Player1.in Player2.in command, and press Enter.
- Result of the program will be seen in the Terminal and a new file called Battleship.out is created in that file location.
- Depending on your choice, either use terminal based programs to display the created file such as vim, nano, gedit or use GUI based programs like NotePad or MousePad to open it. Whole output of the program will be displayed in that file.

RESTRICTIONS ON THE PROGRAM

- The alignment of the ships are assumed. Ships in the given inputs should be either horizontally aligned or vertically aligned.
- If any one of the player has incorrect type of shot, and if that shot prohibits that player the win the game, at the end of the game there will be no winner. In short, if there must be a winner or a draw needs to be happened, inputs should be given accordingly.

APPENDICES

Evaluation	Points	Evaluate Yourself / Guess Grading
Readable Codes and Meaningful Naming	5	5
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	18
There are several negative evaluations	...	...