Name: Halleluyah Woldesenbet Brhanemesqel
Student ID: 20200816

**Practice Assignment #2**

I tested the code with the python 'unittest' module. I have prepared 11-unit tests for the purpose of testing the 2 functions and the 9 object methods we have. I have divided this report based on the unit tests I prepared. Through this unit tests, I was able to find numerous problems on the source code and fix them. I have gone over the fixes I made to the source code when taking about the unit tests I made.

***convert_csv_file_to_objects unit test***

I prepared 2 test cases to test this function. I started by converting the whole 'data.csv' file to a list of country objects using the convert_csv_file_to_objects() function and then chose one country from the list of country objects. For the particular chosen country, I got the player_list attribute. At this point, I divided my test into 2 cases. For the first test case, I traversed the player_list and collected the name of each player in a set. After that, I checked if the player names retrieved matched the expected player names for the country. The returned names though had a problem. Only a single player name was being returned. I went to the source code and tried to see what the problem was. I saw that the convert_csv_file_to_objects() function was assigning a list of only a single player object to the particular country object instead of returning a list of the player objects of the country as specified by the functional requirements. The function was initializing an empty list every time there is a player object that corresponds to a particular country without first checking if a list containing the players already exists. So, I fixed the code by making the function check if a list containing the player objects for a particular country exists before initializing a new list. I have circled in red the part I modified from the source code.

```
def convert_csv_file_to_objects():
    final_players = []

    with open('data.csv', newline='', encoding='utf-8') as csvfile:
        reader = csv.reader(csvfile, delimiter=',', quotechar='|')
        for row in reader:
            final_players.append(row)

    country_objects = []
    country2players = dict()
    for player in final_players:

        if not (player[2] in country2players.keys()):
            country2players[player[2]] = []
        country2players[player[2]].append(
            Player(player[0], player[1], player[2], player[3], player[4]))

    for c, p in country2players.items():
        new_country = Country(c, p)
        country_objects.append(new_country)
    return country_objects
```

Through this test case, I was able to fix the source code and verify that the function converts the CSV file correctly and returns all the players it should.

In my second test case, I chose a particular player from the player_list that was returned and checked if the attribute values of the player matched the expected values. I observed that the type returned for the age and overall value was a string while it should be an integer. The modifications I made for this part are included in the explanation of the 'class Player: def __init__' part since this was an error caused by the problem of another method.

These 2 test cases enabled me to identify the fixes that need to be made to the source code and to confidently fix the errors.

***get_country_object_by_country_name***

The next task I carried out was testing the get_country_object_by_country_name function. I didn't find any errors in this particular function. It was functioning properly as specified by the functional requirement of the code. When given a country name and a list of country objects as input it was properly returning the outputs it should return as specified by the functional requirements. I prepared 2 test cases. The first test case tests if the function returns "No such country" when given a country that isn't included in the list of country objects. The second test case inputs a country name that is expected to be in the list of the country objects. In this test case, I selected the player_list attribute of the returned country object and got the names of the players inside this player_list and checked if the names collected matched the expected player names for the input country. Through this, I verified that they were being returned correctly. So, there was no need for further tests in this function.

***Class Player:***
***def __init__***

In this part, for the purpose of testing the above method I created a new player with all the appropriate information and inserted every information as a string, and checked if the data value and type for each attribute was assigned properly as specified by the functional requirement. I was able to observe that the player's age and player's overall rating were being set as a string. The only error I found on this method was this. So, I went to the source code and made it to convert values

for the player's age and player's overall attributes as integer. I have marked the change I made to the source code by red.

```
def __init__(self, name, age, nationality, overall, position):
    self.name = name
    self.age = int(age)
    self.nationality = nationality
    self.overall = int(overall)
    self.position = position
```

***Class Player:***
***def __gt__***

I tested for this method by preparing two test cases. I created 2 distinct players for each test case. In the first case, I compared 2 players with different overalls. In the second case, I compared 2 players with the same overall but different ages. In this case, I verified that it was returning the player with younger age. I verified this method satisfies every requirement stated on the functional requirement document.

***Class Player:***
***def __str__***

For this method of player, I tested whether the name of the player is returned when using the str() method on the player object or when converting the player object to a string. I didn't find any bugs or errors on the source code on this part. Inside my test function, I created a player object and converted the object to a string using str and verified if it returns the same name as I originally assigned it.

***Class Player:***
***def __repr__***

This method of player should return the name and position of the player as a string separated by a comma in between. To test this, I created a player object and applied the repr() method on the object and verified if it returns a string combination of the name and position of the player separated by a comma. I was able to observe that the original source code was only returning the players name. It was just doing what the __str__ method was doing. So, I fixed the source code to

return the players name and position as a single string separated by a comma. I have included a pic of the changed source code below.

```
def __repr__(self):
    return self.name+", "+self.position
```

***Class Country:***
***def __init__***

In this part, for the purpose of testing the above method I created a new country with all the appropriate information initially inserting every information appropriately as specified by the functional requirement. I was able to observe that the data values and types for the country attributes where set appropriately. My test function tested if the attributes were set as expected.

***Class Country:***
***def get_best_player_and_number_of_world_classes***

I prepared 3 test cases for testing this method. My first test case tested for a country with no players with overall rating of greater than 80. I checked if the method was returning "No world class" as specified by the functional requirement. It wasn't, so I fixed the source code by adding an if condition at the end that checks if best_player value is still 'None'. If it is still 'None', I made the function return a string that says 'No world class'.

```
if best_player==None:
    return 'No world class'
return best_player, cnt
```

My second test case tested a case in which two players with greater than or equal to 80 overall have the same overall. In this case, the method was returning the last player in the list. Instead, it should have returned the first player that appears in the list as the best player. So, I went to the source code and fixed it. I modified the >= sign to the form that is seen in the below picture. I made the best_player name unable to be renamed when a player with equal overall comes later in the list.

```
for player in self.player_list:
    if player.overall >= 80:
        cnt += 1
        if player.overall > best_player_overall:
            best_player = player.name
            best_player_overall = player.overall
```

My third test case tested a case in which there are a large number of players with greater than or equal to 80 overalls, and the best_player doesn't have another player with an equal overall. This test case worked as expected. Through these 3 test cases, I was able to fix and verify the functionality of this method.

***Class Country:***

***def get_best_player_for_position***

For testing this method, I prepared 4 test cases. The first test case was used to check for a case in which the number of players available that satisfy the requirement is equal to the number specified on the input. When I tested the source code using this test case though, a player with the largest overall will replace all the other players on the list and the list will be of the specified size and will have the same player name on all positions. It was rewriting on elements even when there was available space on the list. I fixed the source code so that it would rewrite player names only when there is no None left and also, I made the method sort the list before checking players to rewrite. In this way, I ensured that if there was any player to rewrite the player rewritten would be the first player, the one with list overall score. I have denoted the part I modified by a red pen.

```
def get_best_player_for_position(self, position, number):
    answer = [None] * number

    for player in self.player_list:
        if player.position in POSITIONS[position]:

            for i in range(len(answer)):
                if answer[i] is None:
                    answer[i] = player
                    break
                elif None not in answer:
                    answer.sort()
                    if answer[i]<player:
                        answer[i]=player
                        break
```

The second test case tests a case where the number of players available is less than the number of players specified on the input. In this case our source code was returning a list that contains None as part of the values of the list. Based on the functional requirements specified on the document though, the returned list length should be equal to the players that satisfy the condition. So, I fixed the list size to satisfy the functional requirement. I reduced the size of the list at the end of the method.  I made my code find the index of the first None in the list and cut the list until that point. The returned list as a result won't have the problem we observed. The fix I made is what you see below.

```
if None in answer:
    x=answer.index(None)
    answer=answer[:x]
return answer
```

The third test case checks a case in which there are no players that satisfy the input condition. In this case an empty list was returned. Since the size of the list should match the number of players that satisfy the specified position condition. There are no players that satisfy the condition means that the size of the list should be equal to 0: an empty list.

The fourth test case tests a case in which there are multiple players that have the same overall score and the specified number of players to be chosen is less than the available players in the specific position. Through this we were able to check whether younger players are favored when there are 2 players that have the same overall rating. For the convenience of the person reviewing this paper, I have included all the changes I made to the original function on the pic below.

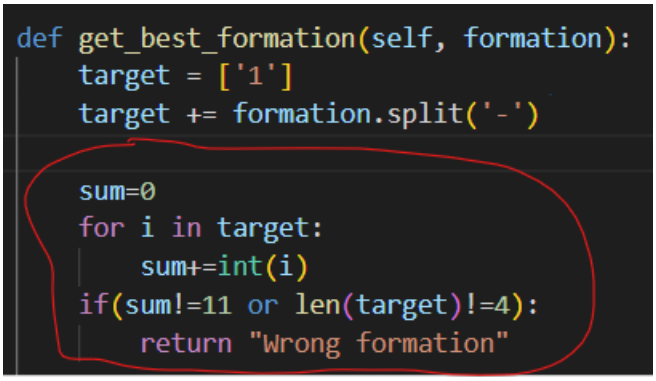
```
def get_best_player_for_position(self, position, number):
    answer = [None] * number

    for player in self.player_list:
        if player.position in POSITIONS[position]:

            for i in range(len(answer)):
                if answer[i] is None:
                    answer[i] = player
                    break
                elif None not in answer:
                    answer.sort()
                    if answer[i]<player:
                        answer[i]=player
                        break
    if None in answer:
        x=answer.index(None)
        answer=answer[:x]
    return answer
```

***Class Country:***
***def get_best_player_for_each_position***

I prepared 2 test cases for testing this method. The first test case checks for a country in which no 2 players have the same overall. Through this method we have been able to observe that the method returns best players for all positions properly. The second one checked a team with the 2 players of the highest overall for a particular position have the same overall values. In this case, the method should choose the player with the least age, and it also behaved in the expected manner.

***Class Country:***
***def get_best_formation***

This was the last unit test I implemented. In this part, I prepared 4 test cases. The first test case checked for a team that has enough number of players that satisfy the given formation. In this case, I also inputted a formation that was valid. When I run this case, the method returned the expected values. My second test case tested for a case in which the formation doesn't fit defender-midfielder-attacker formation, and my third test case tested for a case in which the formation doesn't have 11 players. In both cases as stated on the functional requirements, the method should have returned "Wrong formation", but it didn't. So, I fixed the source code for the method. I wrote a few lines of code within the method that checked if the input formation conforms in terms of the number of players and the formation structure. The additional code I wrote inside the method can be seen below within a red circle like drawing.



```
def get_best_formation(self, formation):
    target = ['1']
    target += formation.split('-')

    sum=0
    for i in target:
        sum+=int(i)
    if(sum!=11 or len(target)!=4):
        return "Wrong formation"
```

In the forth test case, I checked for a team that doesn't have enough players that fit the particular inputted formation. The method should return "Not enough players" as stated on the requirement document, but it wasn't doing that. So, I added an if condition in the source code that checks the

size of the result list formed and returns “Not enough players” when there aren’t enough players to fill the position. I added the method just before the end of the method as seen below.

```
if len(result)<11:
    return "Not enough players"

return result
```

All the changes I made to the source code of this method can be seen below. I have marked the parts I made changes on using a circle.

```
def get_best_formation(self, formation):
    target = ['1']
    target += formation.split('-')
    #I made it check if the formation is valid and if the number of players is enough to fit
    sum=0
    for i in target:
        sum+=int(i)
    if(sum!=11 or len(target)!=4):
        return "Wrong formation"

    result = []
    for i, pos in enumerate(POSITIONS.keys()):
        result += self.get_best_player_for_position(pos, int(target[i]))

    #checking if their aren't enough players for the positon
    if len(result)<11:
        return "Not enough players"

    return result
```

**Conclusion**

I have successfully completed this long journey of preparing tests. I have tried to prepare conclusive tests based on the requirements specified on the document. I prepared unit tests for each method and function. In total, I prepared 11 unit-tests with each unit-test containing a number of test cases to consider. I have given descriptions on how I came up with the test cases I made. Furthermore, I have modified the source code based on the errors I found through my test cases. I have analyzed the results received from the functions and methods, and verified them by the use of my test cases.