**Introduction**

The main objective of this algorithm is who wins the ‘rock paper scissor’ game. The user will play against computer. In this algorithm, user must write which option he wants to choose and then computer will randomly one option from the ‘rock paper scissor’ and it will go through a function to determine who wins the game. The function will return ‘You win, Computer win or Draw’. Whoever wins the game will get one points in his scoreboard. Whoever first to get the 5 points will be the winner of the game. User is allowed to restart the game after the game. To restart the game user, have to choose from the options that will be shown in the terminal. Besides, User is allowed to quit the game anytime he wants, by pressing Q. To input the valid input, user must type ‘rock paper or scissor’. You can have white space before or after the options but not between the option. User must write the full word in order to avoid invalid input.

And then we will have Test Driven Development, to test the algorithm. For our program we have used the Automated unittest unit testing framework to test our program. The unittest frameworkbasically come from Junit (Java) and has similar flavor to large scale unit testing framework in other languages. To test our algorithm, we can give any possible input. Cause, this algorithm can take any input, if its valid it will accept otherwise it will mention that you have typed invalid input. For the TDD, we have to create some test cases and then we have run to verify if everything is working fine. And when we are done with every possible input, we have to run it again to make sure the test case is working.

**Test Cases**

TDD (Test Driven Development) is a cycle that must be done in few steps:

1. Add a test case for TDD
2. Run test cases and verify whether the added test case is succeeded
3. If the test is failed, then we need to modify the algorithm to make the test pass
4. Run the test case again to make sure everything is working fine
5. And then start the new case until its done!

**Test Case - 1**

**Step 1: Add test case**

First, we will add test case ‘paper and scissor’ as we know that whoever choose the scissor is always going to win here. So, we need to add a test case for this part. To run the test case, we need to import the unittest and the object class to run the program. And then we have to create a class in order to get access to the class and then we have to declare a function to check the test case. Example:

import unittest  
from Assignment import Assignment  
class MyTestCase(unittest.TestCase):  
 assignment = Assignment()  
  
 def test\_rock\_paper\_scissor\_equal(self):  
 self.assertEqual('Computer win', self.assignment.play\_game('rock', 'paper'))

In ‘rock paper scissor’ game there is three possible outcomes, you can be win, lose or the game can be draw. So, we cannot use Boolean here, because Boolean does not give us the option to compare three possibilities. So, I use ‘You win, Computer win or Draw’ to verify the result. The function I use it takes two parameters, the first one is user choice and the second one is computer choice, and the functions will return either ‘You win, Computer win or Draw’. In example above the test case is taking ‘rock’ as a user input and ‘paper’ as a computer input. So, the function will return ‘Computer win’ because paper will always win over rock and then my program will check and add the points to the winner.

**Step 2: Run test cases and verify whether the added test case is succeeded**

Let’s say, we want to add another test case, where the user will choose paper and computer will choose scissor. In this sense, computer will win the game, but the one who create the test case mistakenly input as a you win instead of computer win, where the test case will fail.

self.assertEqual('You win', self.assignment.play\_game('paper', 'scissor'))

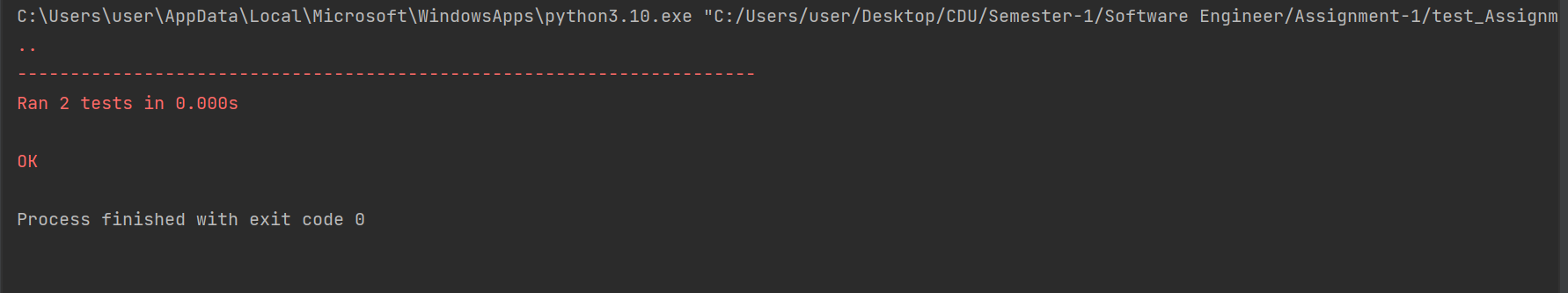
**Step 3: If the test is failed, then we need to modify the algorithm to make the test pass**

Now we have to go through the program and check each statement. If we check the statement in the function, where the user will choose paper and computer will choose scissor then we will come to know that the return value will be computer win. Now we have to change the test case to ‘Computer win’ instead of ‘You win’. Then the test case will pass happily!

self.assertEqual('Computer win', self.assignment.play\_game('paper', 'scissor'))

**Step 4: Run the test case again to make sure everything is working fine**

Until now we have added a new test case, first we get failure as tester did the wrong and then he went through the algorithm and made it work. Now we will run the unittest again to check the output. This time everything should working fine.



**Step 5: And then start the new case until its done!**

Now the tester will go to create a new test case with another possibilities of input and outcome, until the tester is done with every possible outcome.

**Test Case – 2**

**Step 1: Add test case**

First, we will add test case ‘paper and rock’ as we know that whoever choose the paper is always going to win over rock. So, we need to add a test case for this part. To run the test case, we need to import the unittest and the object class to run the program. And then we have to create a class in order to get access to the class and then we have to declare a function to check the test case. Example:

import unittest  
from Assignment import Assignment  
  
  
class MyTestCase(unittest.TestCase):  
 assignment = Assignment()  
  
 def test\_rock\_paper\_scissor\_equal(self):  
   
 self.assertEqual('You win', self.assignment.play\_game('paper', 'rock'))

The function I use it takes two parameters, the first one is user choice and the second one is computer choice, and the functions will return either ‘You win, Computer win or Draw’. In example above the test case is taking ‘paper as a user input and ‘rock as a computer input. So, the function will return ‘You win’ because paper will always win over rock and then my program will check and add the points to the winner.

**Step 2: Run test cases and verify whether the added test case is succeeded**

Let’s say, we want to add another test case, where the user will choose paper and computer will choose rock. In this sense, user will win the game, but the one who create the test case mistakenly input as a computer win instead of You win, where the test case will fail.

def test\_rock\_paper\_scissor\_equal(self):  
  
 self.assertEqual('Computer win', self.assignment.play\_game('paper', 'rock'))

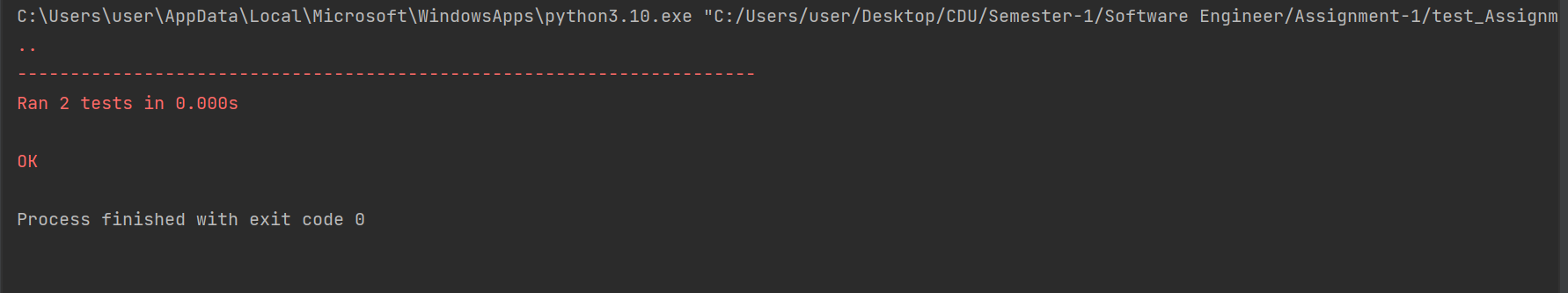
**Step 3: If the test is failed, then we need to modify the algorithm to make the test pass**

Now we have to go through the program and check each statement. If we check the statement in the function, where the user will choose paper and computer will choose rock then we will come to know that the return value will be You win. Now we have to change the test case to ‘You win’ instead of ‘Computer win’. Then the test case will pass happily!

def test\_rock\_paper\_scissor\_equal(self):  
  
 self.assertEqual('You win', self.assignment.play\_game('paper', 'rock'))

**Step 4: Run the test case again to make sure everything is working fine**

Until now we have added a new test case, first we get failure as tester did the wrong and then he went through the algorithm and made it work. Now we will run the unittest again to check the output. This time everything should working fine.



**Step 5: And then start the new case until its done!**

Now the tester will go to create a new test case with another possibilities of input and outcome, until the tester is done with every possible outcome.

**Test case – 3**

**Step 1: Add test case**

First, we will add test case ‘rock and rock as we know that if both are same it is always going to draw. So, we need to add a test case for this part. To run the test case, we need to import the unittest and the object class to run the program. And then we have to create a class in order to get access to the class and then we have to declare a function to check the test case. Example:

class MyTestCase(unittest.TestCase):  
 assignment = Assignment()  
  
 def test\_rock\_paper\_scissor\_equal(self):  
   
 self.assertEqual('Draw', self.assignment.play\_game('rock', 'rock'))

The function I use it takes two parameters, the first one is user choice and the second one is computer choice, and the functions will return either ‘You win, Computer win or Draw’. In example above the test case is taking ‘rock’ as a user input and ‘rock as a computer input. So, the function will return ‘Draw because both choice is same.

**Step 2: Run test cases and verify whether the added test case is succeeded**

Let’s say, we want to add another test case, where the user will choose rock and computer will choose rock. In this sense, computer will win the game, but the one who create the test case mistakenly input as a you win instead of Draw, where the test case will fail.

def test\_rock\_paper\_scissor\_equal(self):  
  
 self.assertEqual('You win', self.assignment.play\_game('rock', 'rock'))

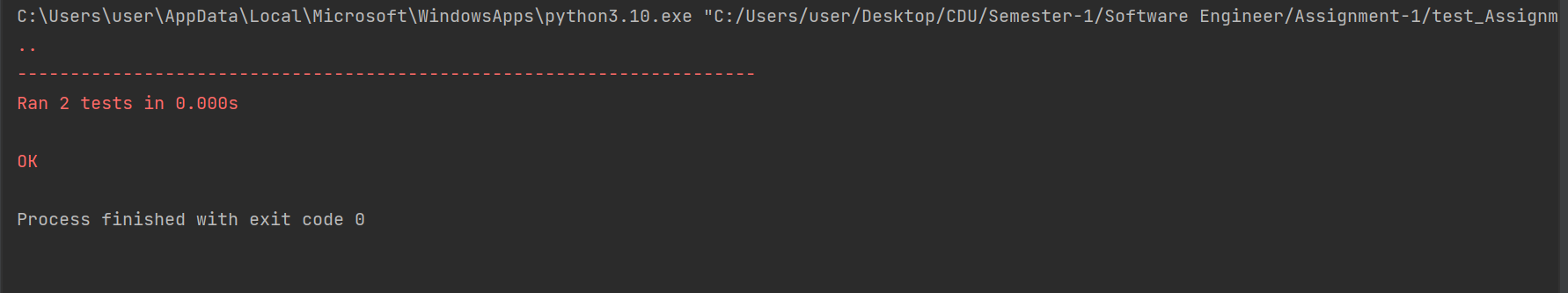
**Step 3: If the test is failed, then we need to modify the algorithm to make the test pass**

Now we have to go through the program and check each statement. If we check the statement in the function, where the user will choose rock and computer will choose rock then we will come to know that the return value will be Draw. Now we have to change the test case to ‘Draw’ instead of ‘You win’. Then the test case will pass happily!

def test\_rock\_paper\_scissor\_equal(self):  
  
 self.assertEqual('Draw', self.assignment.play\_game('rock', 'rock'))

**Step 4: Run the test case again to make sure everything is working fine**

Until now we have added a new test case, first we get failure as tester did the wrong and then he went through the algorithm and made it work. Now we will run the unittest again to check the output. This time everything should working fine.



**Step 5: And then start the new case until its done!**

Now the tester will go to create a new test case with another possibilities of input and outcome, until the tester is done with every possible outcome.

**Conclusion**

There is quit lot of things to learn from this assignment, The main things are How TDD works. After finishing this assignment, I have come to know that - The best benefits from test driven development (TDD) come from continuously improving the code. Test-driven development has benefits beyond just straightforward accuracy confirmation; it can also influence programme design. Organizations utilising a TDD method can easily make changes to their applications without worrying that doing so would "break" the application and hinder their regular operations thanks to the testing modules that are built into the continuous integration development model.

TDD leads to more modularized, flexible & extensible code. This is because the methodology calls for the developers to view of the software as being composed of little pieces that can be created, tested, and then integrated. This results in less rigid and smaller courses.

**Github Link:**

<https://github.com/MdHabib134/Software-Process-and-Tools.git>

**References:**

1. <https://apiumhub.com/tech-blog-barcelona/advantages-of-test-driven-development/>