ROCK, PAPER AND SCISSORS REPORT

INTRODUCTION

Rock, paper and scissors is a hand game usually played between a minimum of two players where each player makes one of three shapes using their hand. The main objective of the task is to write the Rock, paper and scissors game using Test Driven Development (TDD) in python and to outline how the automated unit testing has been used.
The main requirement of the game are listed below:

- The computer should make a random pick among rock, paper and scissors.
- The player should be asked to make a choice among the three options.
- The winner of each round should be with a single point.
- The total of each win or point should be counted and displayed and the first to get a total of 5 points will win the game.
- After a winner is decided there should be an option to quit or restart the game.
- At any time the player should be able to restart the game.

The unit testing framework that has been used is Unittest, which is python's built-in testing framework. Unittest is an automated test tool that verifies small piece of code known as units. These units can be a function or method of a class. Since it runs in an isolated manner the execution time of unit test is very fast. Since unit test check each small pieces of code precisely, they ensure that the program runs properly. Unit test are efficient as they help to reduce the bugs on the code. The code analysis tools used are pylint and flake8. They are used to check the quality of the code.

PROCESS

TDD is the process of software development where we write the test cases first before writing the code first. In this task Unittest has been used to write the test cases. The detailed process of using TDD and unit test is mentioned below.
First the file is created to write the code for the game as the_rps_game.py and the class TheRPSGame is defined. Then test_rps.py file is created for writing the unit test. Then unittest module is imported to the program using:

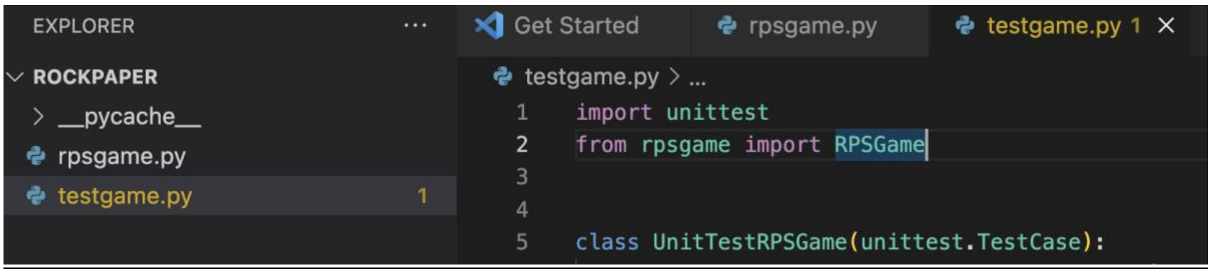
import unittest.

The testgame.py then access the RPSGame class by importing it from rpsgame.py as

import unittest
from rpsgame import RPSGame

The next step will be to create a test case which is done by defining a new class UnitTestRPSGame that will inherit the test case from unittest.

class UnitTestRPSGame(unittest.TestCase):



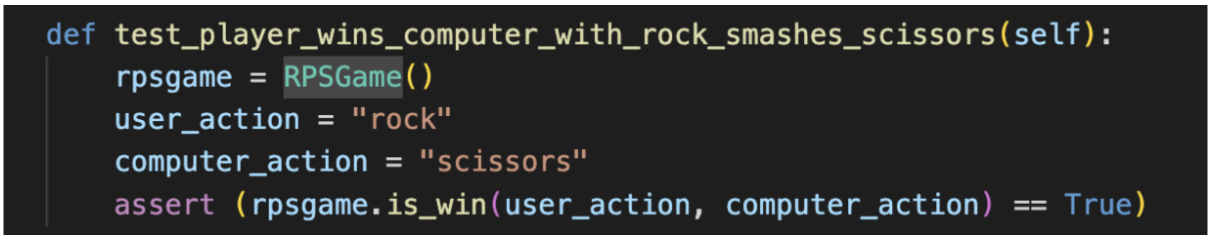
```
import unittest
from rpsgame import RPSGame


class UnitTestRPSGame(unittest.TestCase):
```

Then we define a method showing the case where rock wins the scissors and to test this method we add a method call as

def test_player_wins_computer_with_rock_smashes_scissors(self):

In this process, we create a new instance of the class RPSGame. Then the user_action has the value rock, which is the user input. The computer_action is set to scissors. Then the assert section is used to check whether the code return the expected value. In this case the winner is the player who selects the rocks and wins over the scissors and hence the user win count will be True.

assert (rpsgame.is_win(user_action, computer_action) == True)



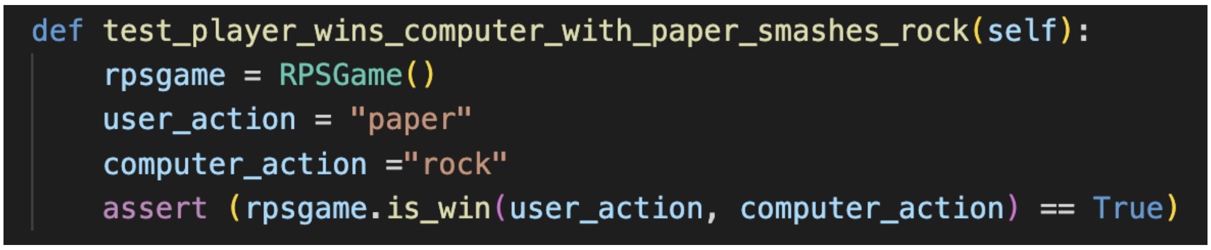
```
def test_player_wins_computer_with_rock_smashes_scissors(self):
    rpsgame = RPSGame()
    user_action = "rock"
    computer_action = "scissors"
    assert (rpsgame.is_win(user_action, computer_action) == True)
```

Next, we define the test case as below:

def test_player_wins_computer_with_paper_smashes_rock(self):

In this test case, the user_action has the value paper, which is the user input. The computer_action is set to rock. Then the assert section is used to check whether the code return the expected value. In this case the winner is the player who selects the paper and wins over the rock and hence the user win count will be True.

assert (rpsgame.is_win(user_action, computer_action) == True)



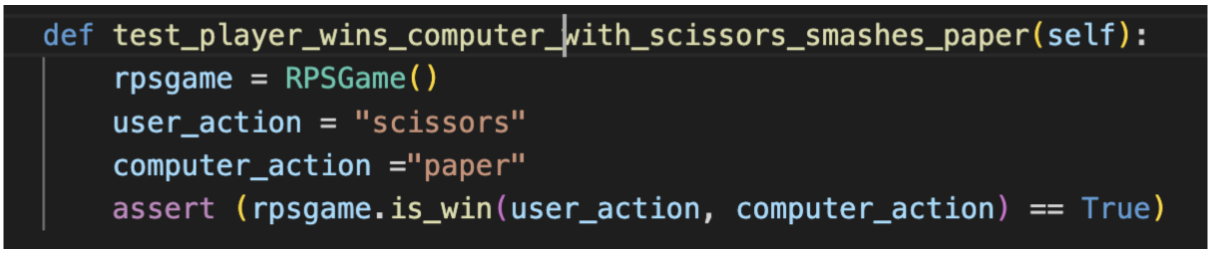
```
def test_player_wins_computer_with_paper_smashes_rock(self):
    rpsgame = RPSGame()
    user_action = "paper"
    computer_action ="rock"
    assert (rpsgame.is_win(user_action, computer_action) == True)
```

Furthermore, we define the test case as below:

def test_player_wins_computer_with_scissors_smashes_paper(self):

In this test case, the user_action has the value scissors, which is the user input. The computer_action is set to paper. Then the assert section is used to check whether the code return the expected value. In this case, the winner is the player who selects the scissors and wins over the paper and hence the user win count will be True.

assert (rpsgame.is_win(user_action, computer_action) == True)



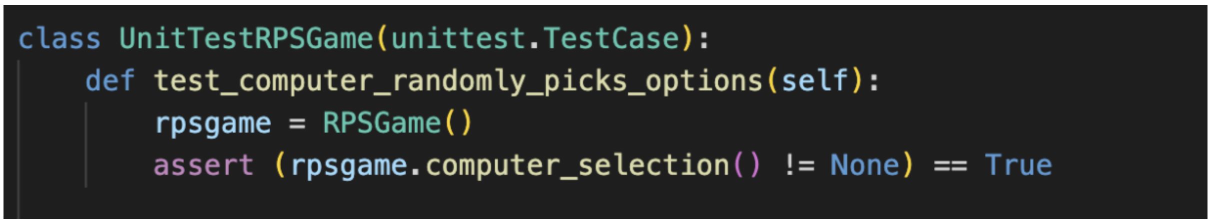
```
def test_player_wins_computer_with_scissors_smashes_paper(self):
    rpsgame = RPSGame()
    user_action = "scissors"
    computer_action ="paper"
    assert (rpsgame.is_win(user_action, computer_action) == True)
```

Another test case that has been shown is the testing showing the random pick done by the computer which is shown below:

def test_computer_randomly_picks_options(self):

In this test case, we check whether the computer has picked a random choice or not. If the random choices in not equal to NULL, that means the computer has made a random pick.

assert (rpsgame.computer_selection() != None) == True

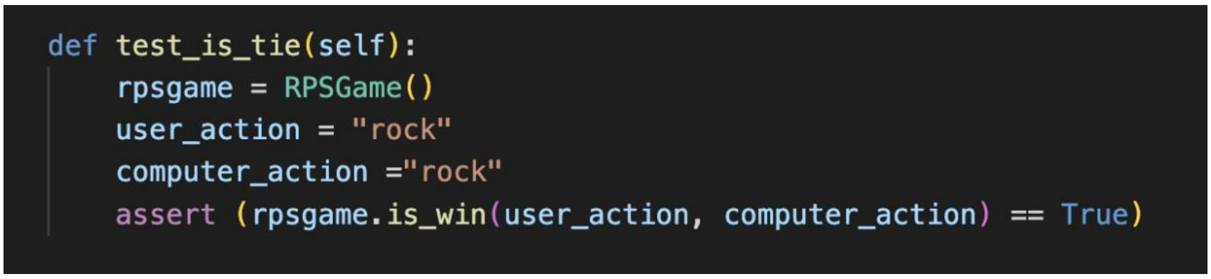


```python
class UnitTestRPSGame(unittest.TestCase):
    def test_computer_randomly_picks_options(self):
        rpsgame = RPSGame()
        assert (rpsgame.computer_selection() != None) == True
```

The next test case is for testing the final winner of the game. The player that wins the game for five times in total becomes the winner. When each round is over the total win count of the winning team is added. The method of test case is as given below:

```
def test_is_tie(self):
```

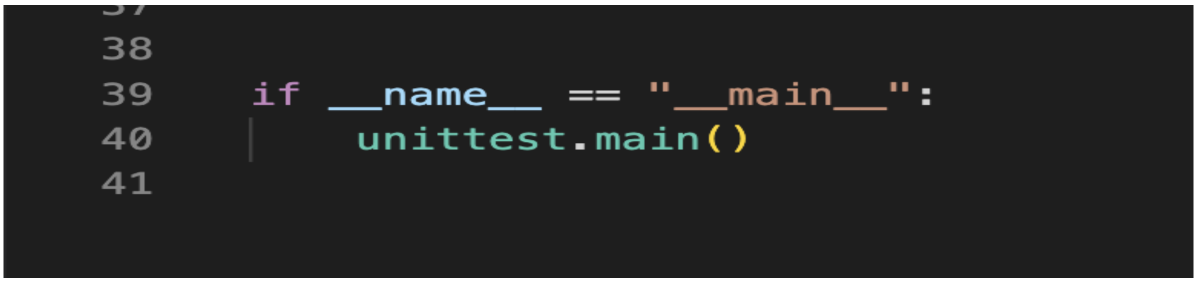
In this test case, we test for the condition where both player and the computer chose the same option which results in a tie. Here we check for the condition where the user_action is equal to computer_action where both of them are assigned to the same option from rock, paper and scissors.



```python
def test_is_tie(self):
    rpsgame = RPSGame()
    user_action = "rock"
    computer_action ="rock"
    assert (rpsgame.is_win(user_action, computer_action) == True)
```

Before running the test, we call the main() function of the unittest module as follows:

```
if __name__ == “__main__”:
        unittest.main()
```

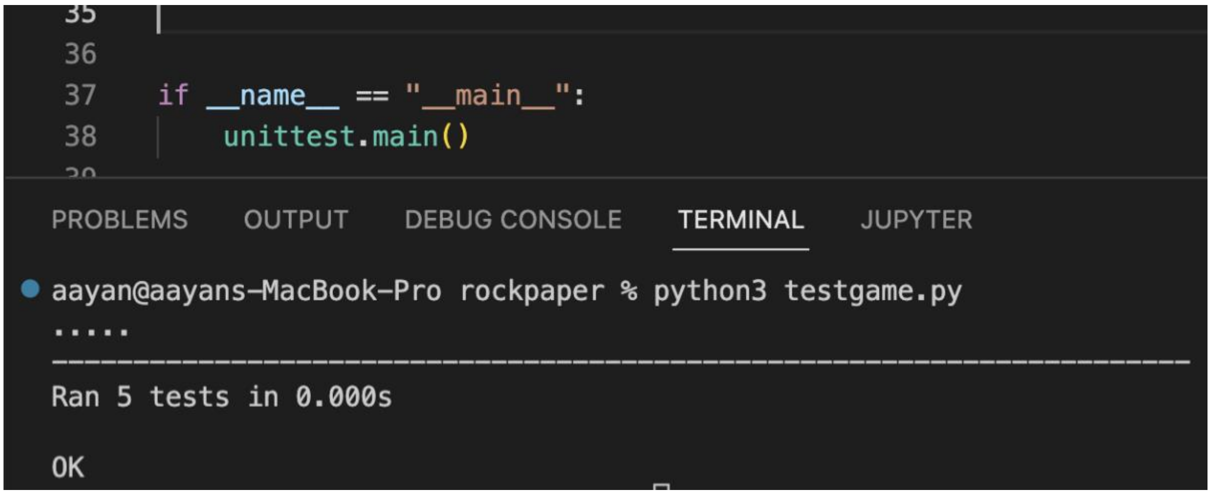


```python
if __name__ == "__main__":
    unittest.main()
```

After this we use the terminal to run the test by navigating to the folder and executing the command below:

```
python3 testgame.py
```

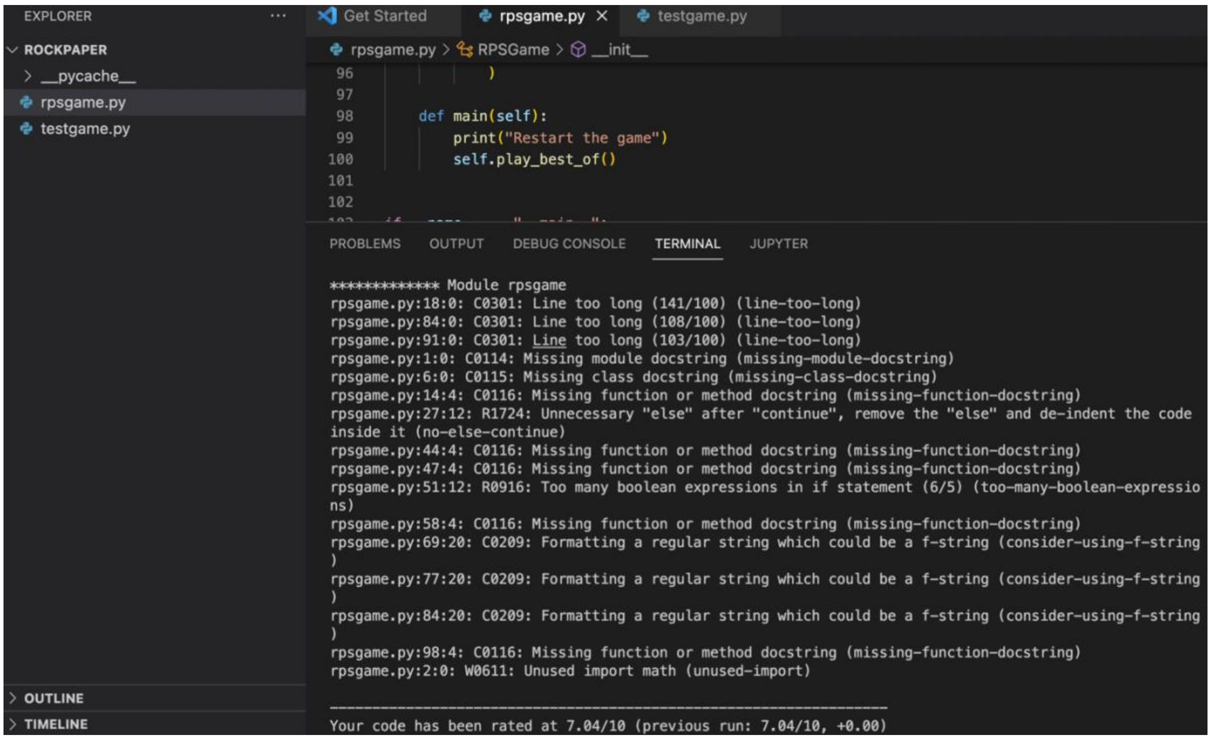
The output for the above command will be:



```
if __name__ == "__main__":
    unittest.main()
```

```
aayan@aayans-MacBook-Pro rockpaper % python3 testgame.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

In this task, code analysis tool such as pylint and flake8 was also used. These tools helped to show the quality of the code. Also, these tools provide the rating of our code and we can see the remarks that could help to improve the quality of the code. The pylint can be executed by using the terminal.
In the terminal, we need to use the command as pylint rpsgame.py. The tool flake8 can also be used to see how efficient the code is. It can be used in terminal by using the command as flake8 rpsgame.py.



```
            )

    def main(self):
        print("Restart the game")
        self.play_best_of()
```

```
************* Module rpsgame
rpsgame.py:18:0: C0301: Line too long (141/100) (line-too-long)
rpsgame.py:84:0: C0301: Line too long (108/100) (line-too-long)
rpsgame.py:91:0: C0301: Line too long (103/100) (line-too-long)
rpsgame.py:1:0: C0114: Missing module docstring (missing-module-docstring)
rpsgame.py:6:0: C0115: Missing class docstring (missing-class-docstring)
rpsgame.py:14:4: C0116: Missing function or method docstring (missing-function-docstring)
rpsgame.py:27:12: R1724: Unnecessary "else" after "continue", remove the "else" and de-indent the code
inside it (no-else-continue)
rpsgame.py:44:4: C0116: Missing function or method docstring (missing-function-docstring)
rpsgame.py:47:4: C0116: Missing function or method docstring (missing-function-docstring)
rpsgame.py:51:12: R0916: Too many boolean expressions in if statement (6/5) (too-many-boolean-expressio
ns)
rpsgame.py:58:4: C0116: Missing function or method docstring (missing-function-docstring)
rpsgame.py:69:20: C0209: Formatting a regular string which could be a f-string (consider-using-f-string
)
rpsgame.py:77:20: C0209: Formatting a regular string which could be a f-string (consider-using-f-string
)
rpsgame.py:84:20: C0209: Formatting a regular string which could be a f-string (consider-using-f-string
)
rpsgame.py:98:4: C0116: Missing function or method docstring (missing-function-docstring)
rpsgame.py:2:0: W0611: Unused import math (unused-import)

------------------------------------------------------------------
Your code has been rated at 7.04/10 (previous run: 7.04/10, +0.00)
```

CONCLUSION

From this rock, paper, scissors game which was written using TDD and unit testing, we can see that there are fewer chances of errors in the code. Writing the test cases at the beginning will bring improvement in the codes and that will help to run the program effectively.
The things that we learnt from this task are listed below:
i. TDD and Unit Testing made it easier to detect and reduce the bug.
ii. We could easily write the main code for the game after using the test cases and that increased our efficiency.

ii. Using this approach will help to track the software development by repeatedly testing the software with several test cases
iii. TDD helps to save time and reduce efficiency by minimizing the time to debug the errors.
iv. Unit testing helps to increase the developer’s productivity.
There are certain fields where there need improvements that we learned from this task.
i. Only two players: the user and the computer were playing the game. The can be played by more players at the same time.
ii. This approach lacks proper testing of the non-functional attributes of a program.
iii. Also, the unit testing is not quite efficient in catching the complex errors in the system ranging from multiple modules. If these things can be improved, then unit testing can be more reliable and efficient.
The things that can be done to improve TDD and unit testing is listed below:
i. Writing a clean code will help to make the test cases more effective.
ii. Refactoring the codes while keeping the test passing can increase efficiency.

The GitHub Link for the task is given below:
https://github.com/Ram8510/assignment_1_SE.git