Viktor Tiger

Software Unit Testing Report

PRT582

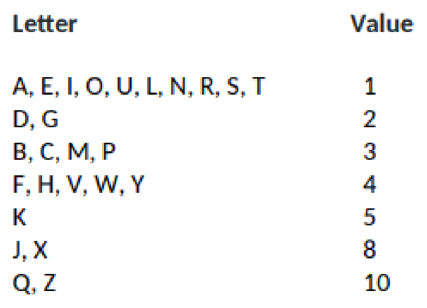
Introduction

This project aims to develop a Scrabble scoring game using test driven development (TDD). TDD is a coding methodology where unit tests are created before the implementation of the program. This ensures that the code produces expected results and handles edge cases. Alongside TDD, linting tools are used to quality assure the code.

The program is developed using Python, a programming language useful for small application due to the different libraries available and the small amount of code needed to create a functioning application. To perform the unit testing needed the library “unittest” was used. It provides functionality to easily set up automated test cases for all parts of the program. To perform linting the libraries “flake8” and “pylint” were used. “Flake8” identifies logical errors for example extra blank spaces or too long lines of code. “Pylint” runs a more comprehensive check to ensure the code is up to standard. It gives feedback on missing function descriptions, naming standard and much more.

The requirements on the game are as follows:

- The user has fifteen seconds to input an answer.
- Input is not case sensitive.
- Input is a valid English word.
- The game is ten rounds long or until the user chooses to quit.
- Scoring is calculated using the table in figure 1.



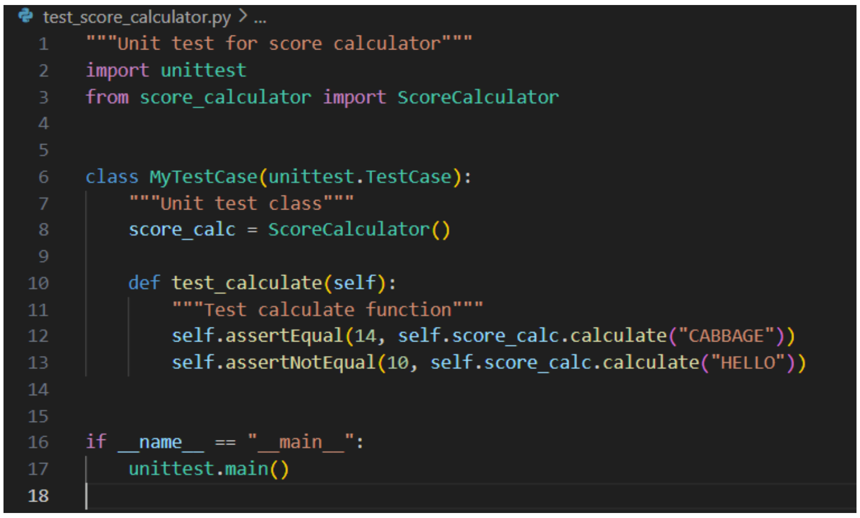
Letter	Value
A, E, I, O, U, L, N, R, S, T	1
D, G	2
B, C, M, P	3
F, H, V, W, Y	4
K	5
J, X	8
Q, Z	10

Figure 1. Scrabble score table

Process

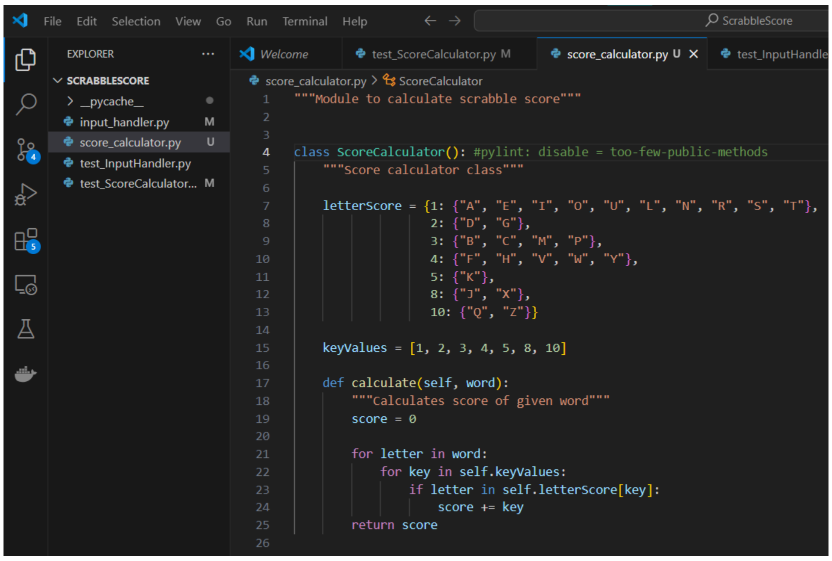
The process of following TDD is straightforward. It begins with writing the test cases, as seen in figure 2. The test is designed to verify the score calculation functionality. Running the tests will result in failure due to the “scoreCalculator” class not being implemented yet.

The next step is to create the “ScoreCalculator” class and its functions. During this phase, the tests are continuously run to get instant feedback if they work as intended or not. This process continues until all tests passes.



```python
"""Unit test for score calculator"""
import unittest
from score_calculator import ScoreCalculator


class MyTestCase(unittest.TestCase):
    """Unit test class"""
    score_calc = ScoreCalculator()

    def test_calculate(self):
        """Test calculate function"""
        self.assertEqual(14, self.score_calc.calculate("CABBAGE"))
        self.assertNotEqual(10, self.score_calc.calculate("HELLO"))


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

```

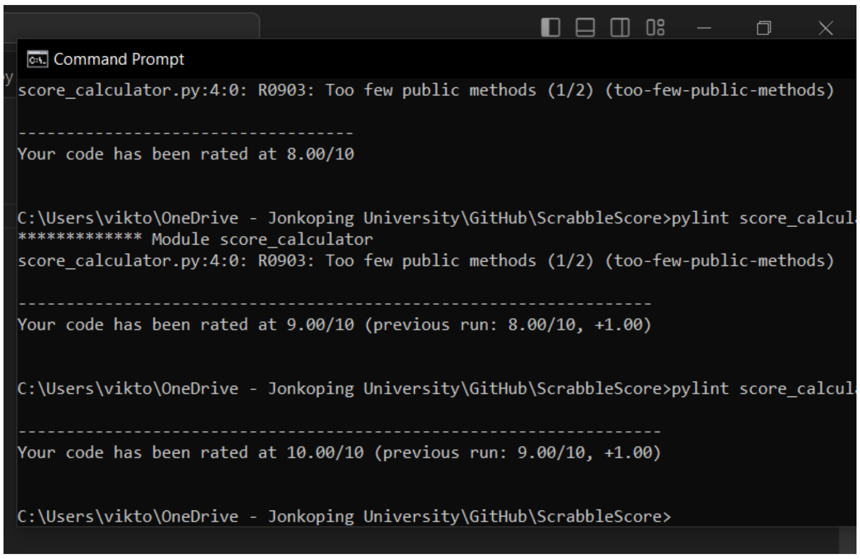
Figure 2. Unit test for score calculator



```python
"""Module to calculate scrabble score"""


class ScoreCalculator(): #pylint: disable = too-few-public-methods
    """Score calculator class"""

    letterScore = {1: {"A", "E", "I", "O", "U", "L", "N", "R", "S", "T"},
                   2: {"D", "G"},
                   3: {"B", "C", "M", "P"},
                   4: {"F", "H", "V", "W", "Y"},
                   5: {"K"},
                   8: {"J", "X"},
                   10: {"Q", "Z"}}

    keyValues = [1, 2, 3, 4, 5, 8, 10]

    def calculate(self, word):
        """Calculates score of given word"""
        score = 0

        for letter in word:
            for key in self.keyValues:
                if letter in self.letterScore[key]:
                    score += key
        return score

```

Figure 3. Score calculator class

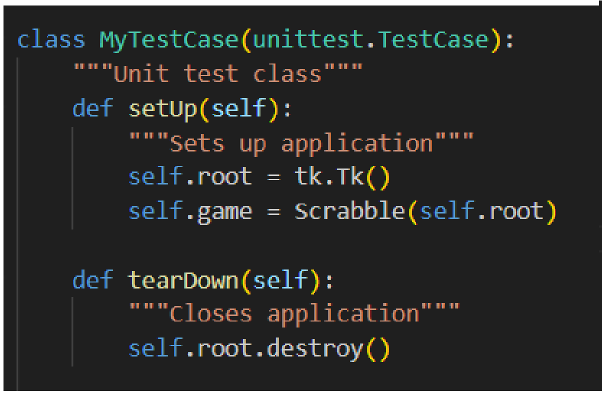
Once all tests have passed, the code undergoes a quality check using “Flake8” and “pylint” as seen in figure 4. If any warnings or issues are identified then the code is revised to meet the required standard. When both “flake8” and “pylint” don’t give any warnings the code is re-run against the unit tests to ensure no detrimental changes were made. Only after all these steps the code is pushed to the main branch on Git. This process is then repeated when creating all other modules.



```
score_calculator.py:4:0: R0903: Too few public methods (1/2) (too-few-public-methods)

-----------------------------------
Your code has been rated at 8.00/10


C:\Users\vikto\OneDrive - Jonkoping University\GitHub\ScrabbleScore>pylint score_calcul
************* Module score_calculator
score_calculator.py:4:0: R0903: Too few public methods (1/2) (too-few-public-methods)

------------------------------------------------------------------
Your code has been rated at 9.00/10 (previous run: 8.00/10, +1.00)


C:\Users\vikto\OneDrive - Jonkoping University\GitHub\ScrabbleScore>pylint score_calcul

-------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 9.00/10, +1.00)


C:\Users\vikto\OneDrive - Jonkoping University\GitHub\ScrabbleScore>
```

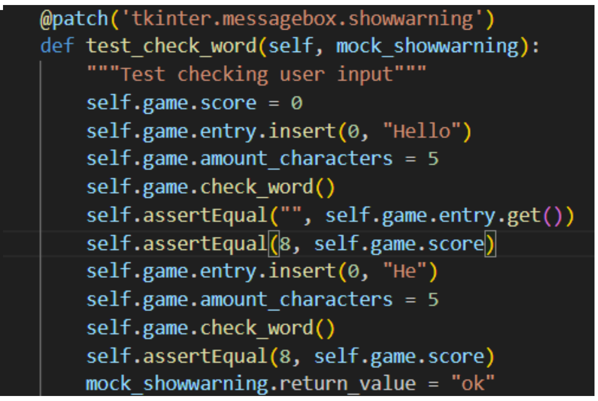
Figure 4. Linting result for score calculator

Unit testing an application is a bit different. The test case needs extra functions for set up and tear down of the test, as seen in figure 5. When testing an application there is a need to mimic user input. This is done with patch and insert. A string gets inserted into the input field via insert and patch simulates the user pressing ok on expected "messagebox" pop up, as seen in figure 6.



```python
class MyTestCase(unittest.TestCase):
    """Unit test class"""
    def setUp(self):
        """Sets up application"""
        self.root = tk.Tk()
        self.game = Scrabble(self.root)

    def tearDown(self):
        """Closes application"""
        self.root.destroy()
```

Figure 5. Set up and tear down function



```python
@patch('tkinter.messagebox.showwarning')
def test_check_word(self, mock_showwarning):
    """Test checking user input"""
    self.game.score = 0
    self.game.entry.insert(0, "Hello")
    self.game.amount_characters = 5
    self.game.check_word()
    self.assertEqual("", self.game.entry.get())
    self.assertEqual(8, self.game.score)
    self.game.entry.insert(0, "He")
    self.game.amount_characters = 5
    self.game.check_word()
    self.assertEqual(8, self.game.score)
    mock_showwarning.return_value = "ok"
```

Figure 6. Patched unit test

Conclusion

The linting tools are a great way to easily improve code quality. Over the course of this project my coding changed to match the tools standard which reduced the time needed to fix the code. By writing unit tests first more time was spent on planning what functions and classes were needed. This minimized time spent on writing the program. Lastly TDD is great due to the guarantee that the code works as intended and ease to test after making substantial changes.

Challenges with TDD includes difficulty testing the graphical user interface (GUI) and timed functionality. Unit tests are by design meant to be fast to execute which makes it poor standard and difficult to force it to wait. The game has a timer where the user has fifteen seconds to answer, the timer functionality could not be unit tested.

In part due to poor planning of the GUI and it’s more complex structure it was hard to write good unit tests ahead of time for the GUI. The GUI unit tests were rewritten a few times and added upon to match the needs of the GUI. In the future a solution is to draw a mock-up of the application and a user flow chart. With improved planning all unit tests will be easier to write.

Github repository: https://github.com/vTigerJU/ScrabbleScore