PRT582

SOFTWARE UNIT TESTING REPORT

Prashil Koirala

S366956

Table of Contents

Table of Screenshots:

Introduction

In this report, I describe how I used TDD in python to create a number guessing game. Firstly, I will outline the requirements of the game:

1. The game randomly generates a 4-digit number.
2. The program will keep asking the user to guess the number until the player guesses it correctly or has quitted.
3. The program will give hints to the used based on their inputs. An X means a digit is in the wrong spot, a circle means a digit is in the right spot and _ means the random number does not contain the digit.
4. Once the game is over, the program will display in how many attempts the user won the game.
5. The user can quit anytime by pressing enter.
6. The program asks if the user wants to play again after they win.

I will be using 'unit test' module of python as the testing tool.

unittest is a built-in testing framework in Python, designed to help developers write and run unit tests for their code. Unit testing is an essential practice in software development that involves testing individual parts (units) of a program to ensure they work correctly in isolation.

The unittest framework provides a structured and organized way to create and manage test cases, making it easier to validate the correctness of your code as you make changes. It encourages modular code design and promotes good coding practices by separating testing logic from your actual codebase.

Here's a brief overview of how unittest works:

1. Test Case Class: In unittest, you define test cases as classes that inherit from the unittest. TestCase class. Each test case class contains methods that represent different tests.
2. Test Methods: Test methods within a test case class are named starting with the word "test". These methods contain the code that exercises the specific functionality you want to test.

3. Assertions: Inside the test methods, you use various assertion methods provided by unittest to check whether the expected behavior of your code matches the actual behavior. Common assertion methods include assertEqual, assertNotEqual, assertTrue, assertFalse, and so on.

4. Test Discovery and Execution: unittest provides a test discovery mechanism that automatically locates test cases and methods in your codebase. You can run your tests using the built-in test runner by invoking the unittest.main() function or using test runners provided by test integration tools.

Process

I started by writing the first simple test case for my program. It was to check whether the game generates a random number or not.

```
class MyTestCase (unittest.TestCase):
    '''Class for unit test'''
    game.is_from_test=True
    random.seed(52) #seed to produce the same number making it easier to test
    num= game.number_generator() #call the actual function

    def test_number_generator(self):
        '''This function tests if the number generator is generating numbers'''
        print(self.num)
        self.assertIsNotNone(self.num)
        self.assertEqual(self.num,5396)
```

Figure 1: First Unit test code for random number generation

I used a seed so I can check whether the random number is equal every time I run the test. This test failed in the beginning. Then I wrote the code to make it pass. Then, I wrote the test to see if the user wanted to quit the game. The user would have to press enter to quit. Any other input meant the user was not quitting.

```
def test_check_if_user_is_quitting(self):
    '''This function tests if the user is quitting'''
    input1=''
    input2='1'
    output1 = game.check_if_user_is_quitting(input1)
    output2= game.check_if_user_is_quitting(input2)
    self.assertEqual(output1, True)
    self.assertEqual(output2, False)
```

Figure 2: Test case for user quitting

The next step in my program was to write a test to test the validity of the inputs. Getting user input for every run of the test was not feasible, so I wrote the test and program in such a way that I could give hard-coded inputs for the program to test. Only 4-digit numbers are treated as valid inputs by the program.

```
def test_check_if_valid_input(self):
    '''This functions tests if the input is valid'''
    input1='a'
    input2='!!'
    input3='111'
    input4='1235'
    input5='7281'
    output1 = game.check_if_valid_input(input1)
    output2= game.check_if_valid_input(input2)
    output3 = game.check_if_valid_input(input3)
    output4= game.check_if_valid_input(input4)
    output5= game.check_if_valid_input(input5)
    self.assertEqual(output1, False)
    self.assertEqual(output2, False)
    self.assertEqual(output3, False)
    self.assertEqual(output4, True)
    self.assertEqual(output5, True)
```

Figure 3: Test case for checking input validity

After the test failed, I wrote the code for the actual game to check input validity.

```
def check_if_valid_input(input_string):
    """This functions checks if the entered input is valid for our game"""
    if re.search("[a-z]", input_string.lower()):
        return False
    try:
        integer_guess = int(input_string)
        if (
            integer_guess < 1000 or integer_guess > 9999
        ):  # if the input is 3 digit or 5+ digits
            return False
        return True
    except ValueError:
        return False
```

Figure 4: Code to check input validity

Any alphabets were immediately invalid. With a try and except block, I raised an error for any input that was not an integer. In this way, I handled the inputs.

With all this done, the next step was to guess the number and provide hints to the user. I wrote a test case first with hardcoded inputs and hardcoded number to be guessed.

```
def test_guessing_hints(self):
    '''This functions tests if the input is equal to the generated random number'''
    #Our number is 5396 with the seed
    input1='1234'
    input2='7286'
    input3 = '4821'
    input4 = '9276'
    input5 = '9536'
    input6 = '5396'

    game_output1 = game.check_if_correct_guess(input1)[1]
    game_output2 = game.check_if_correct_guess(input2)[1]
    game_output3 = game.check_if_correct_guess(input3)[1]
    game_output4 = game.check_if_correct_guess(input4)[1]
    game_output5 = game.check_if_correct_guess(input5)[1]
    game_output6 = game.check_if_correct_guess(input6)[1]

    # self.assertEqual(game_output1, ["_","circle","_","_"])
    self.assertEqual(game_output1, ["_", "_", "x", "_"])
    self.assertEqual(game_output2, ["_", "_", "_", "circle"])
    self.assertEqual(game_output3, ["_", "_", "_" ,"_"])
    self.assertEqual(game_output4, ["x", "_", "_", "circle"])
    self.assertEqual(game_output5, ["x", "x", "x", "circle"])
    self.assertEqual(game_output6, ["circle", "circle", "circle", "circle"])
```

Figure 5: Test case for providing hints

I decided that I would be returning a list of a boolean and the list of hints from the function that checks the input against the random generated number. In the test, the random number was set to 5396. So, using the assertEqual method, I was able to expect what output should be returned for

each of the inputs. The tests failed and then, I coded the function.

```
def check_if_correct_guess(input_string):
    """This function provides the hints for the guesses."""
    num_list = []
    num_list2 = []
    hint_list = ["_", "_", "_", "_"]
    manipulating_random_number = NUM
    mainpulating_user_input = int(input_string)

    for i in range(0, 4):
        #splitting the 4 digit number into single digits by using remainders after dividig by 10.
        remainder1 = manipulating_random_number % 10
        remainder2 = mainpulating_user_input % 10
        manipulating_random_number = manipulating_random_number // 10
        mainpulating_user_input = mainpulating_user_input // 10
        num_list.append(remainder1)
        num_list2.append(remainder2)
    num_list.reverse() #reversing to get the actual number again
    num_list2.reverse()
    for i,element in enumerate(num_list2):
        element = num_list2[i]
        if element in num_list:
            if num_list[i] == element:
                hint_list[i] = num_list2[i] = "circle"
            else:
                hint_list[i] = num_list2[i] = "x"

    print("HINTS: ", hint_list)

    if hint_list.count("circle") == 4:
        return [True, hint_list]
    return [False, hint_list]
```

Figure 6: Function to provide hints based on input

After this the tests passed and I had a fully functioning game. Then I used pylint to obtain a 10/10 score on my code for both game.py and game_test.py.

Conclusion

I learnt that with TDD, it leaves very less margin for error when it comes to unit testing. I thought of all the scenarios that need to be handled by my code before actually writing the code. Then, I made test cases, which failed. After I wrote the codes to make the tests pass, my code was basically error free as all the possible errors were handled during the process of making the tests pass. I finally used GitHub to upload the codes. The repo can be found at:
https://github.com/prashilthegreat/PRT582_366956

Lessons Learnt:

1. Unit testing a program with inputs can be difficult. I spent a lot of time figuring out how to hard code the inputs instead of the program asking for input during the tests.

```python
def main(is_from_test=False):
    """This is the main function"""
    if (
        not is_from_test
    ):  # in unit test im giving hard coded inputs, different test cases
        get_user_input()


if __name__ == "__main__":
    NUM = number_generator()
    main()
else:
    NUM = 5396
    main(True)
```

   This is how I handled that problem. If the program is running from the main file, the program asks for inputs. If the program is running from the test file, no input is required. This makes is easier to test 100s of numbers without ever asking for input.

2. Without complete practice in TDD, I was going back and forth between writing test cases then coding and sometimes coding first and fitting the test cases to my code. This can be improved upon, and I will be much more efficient with TDD for the future.