Development of "Guess the Number" Game

Introduction:

The game "Guess the Number" provides players with an enjoyable and engaging means of improving their logical abilities. The game's objective is to allow players to guess the system's randomly generated four-digit numbers according to the game prompts. As players make guesses, the game offers guidance in "o" and "x" to help them arrive at the correct number. For example, the "o" indicates that the player correctly guessed the number and position. An "x" means the number is valid but not perfectly positioned. If the player guesses wrong, the game continues until the player successfully guesses the number or chooses to quit. At the end of the game, players are told how many times they tried.

To ensure the game's integrity, quality and reliability, we adopted a test-driven development (TDD) approach. At the same time, we use unittest as automated testing tools to test the code. Unittest is an open-source framework for writing and running repeatable automated tests. It allows us to systematically test individual units of the game's source code to determine whether each part functions as expected. At the same time, we use unittest with tools such as coverage.py, which can help us improve test coverage.

Our purpose in using automated testing tool is to verify the implementation of the main functions of the game, such as random number generation, interpretation of player guesses, and providing accurate hints. Evaluate how the game responds in various situations, such as unexpected input or midway quit. At the same time, achieve as high test coverage as possible, ensuring that most code paths are tested and verified.

All in all, combining the interesting interactive mode of the "guess the number" game and the rigorous testing framework provided by unittest will give users a pleasant gaming experience.

Process:

Test Driven Development (TDD) is a software development methodology. It requires writing the test code before writing the code of a specific function and then only writing the function code that makes the test pass and promotes the whole development through the test. A typical implementation of the TDD process is to write tests, write code, and refactor.

Firstly, we write the failing test by using the unit testing framework in Python. Before writing the specific functionality of the number-guessing game, we first write tests for the desired functionality. The test will likely fail since we have yet to write the functionality.

1. Write a Failing Test

First requirement: Randomly generate a four-digit number

Before implementing random number generation for a "guess the number" game, we might write a test to ensure the generated numbers are four digits long. And it is guaranteed that the number range is between 0000 and 9999.

```python
import unittest
from game import Game

class TestGame(unittest.TestCase):

    def test_player_number(self):
        game = Game()
        number = game.player_number()
        self.assertEqual(len(number), 4)
        self.assertTrue(1000 <= int(number) <= 9999)
```

Picture1.1: unittest for four-digit number

Second requirement: the player will be asked to quit or to play again.

Before each game starts, the program will ask the player whether to continue guessing the number game(Y/N). If the player answers ‘N’, the system will automatically exit the game.

```python
    def test_quit_game(self):
        game = Game()
        self.assertTrue(game.quit_game('N'))
        self.assertTrue(game.quit_game('N'))
        self.assertFalse(game.quit_game('1234'))
```

Picture 1.2: unittest for ask to quit

Third requirement: As the player makes guesses, the game provides assistance through the use of "o" and "x" symbols to aid in reaching the correct number.

After players enter the number, the program will respond with hints using ‘o’ and ‘x’. A ‘o’ indicates that one digit is correct and is in the right spot. 'x' indicates that one digit is correct but in the wrong spot.

```python
    def test_check_guess(self):
        game = Game()
        game.number = "0987"
        self.assertEqual(game.guess("0977"), "ooxo")
        self.assertEqual(game.guess("4321"), "")
        self.assertEqual(game.guess("1243"), "")
        self.assertEqual(game.guess("9078"), "xxxx")
```

Picture 1.3: unittest for hint

Fourth requirement: Player can quit the game anytime.

```python

    @patch('builtins.input', return_value='N')
    def test_game_ends_on_quit(self, mock_input):
        game = Game()
        game.play()
```

Picture 1.4: unittest for quit

2. Write the Minimal Code to Make the Test Pass

Secondly, we wrote the minimal code necessary to make the test pass. We only write the necessary amount of code to pass the test, even if it's not flawless or complete, as long as it meets the requirements of the test.

For the test of the first requirement, we only need to ensure that it is a four-digit number. Then the test will be passed.

```
def player_number(self):
    return '1234'
```

Picture 2.1: minimal code for four-digit number

For the second requirement test, we only need to ensure that the input N returns True, and the rest of the inputs return False. Then we will pass the test.

```
def quit_game(self,player_input):
    if player_input == "N":
        return True
    return False
```

Picture 2.2: minimal code for ask to quit

In order to meet the third requirement. We are also looping through the number guessed by the user and the correct number. When the number and position are correct, use ‘o’ to indicate. Use an 'x' when the number is correct but not in the correct position.

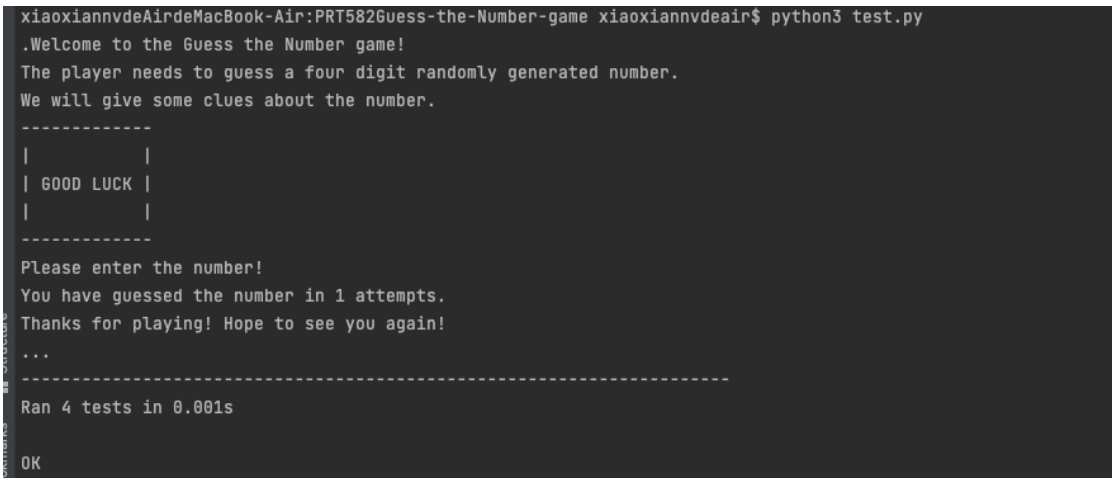
```
def guess(self, guess):
    answer = []
    for i in range(4):
        if guess[i] == self.number[i]:
            answer.append("o")
        elif guess[i] in self.number:
            answer.append("x")

    return ''.join(answer)
```

Picture 2.3: minimal code for hint

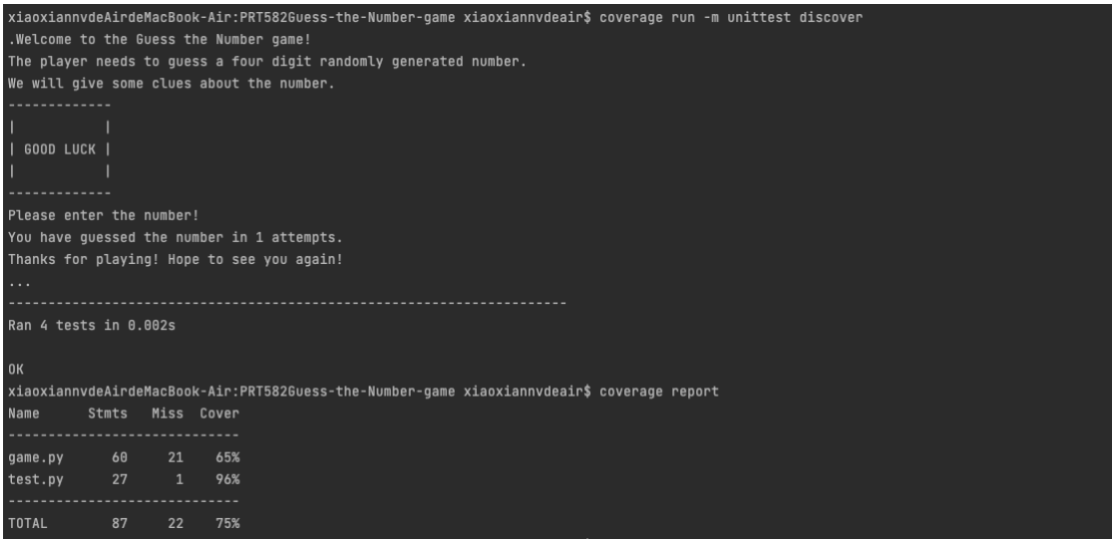
3. Refactor

With the tests passing, we can now safely refactor our code, ensuring optimizations without affecting its functionality while ensuring our tests still pass. Once all tests pass, the refactoring process ends. This step may involve optimizing the code or refactoring it for better readability. After the refactoring step, we return to the first step and write new tests for the next feature or bug fix. Then, keep letting it go and refactor again. This cycle is repeated for each game requirement to ensure a robust and functional implementation.

Ultimately, we completed the number-guessing game through Test Driven Development (TDD). The code passes all known test cases. With unittest and coverage, we get a test coverage of 75%.



Picture 3.1: pass all the test case



Picture 3.2: Test coverage report

Conclusion:

Throughout the process of developing the "guess the number" game using Python Test Driven Development (TDD), we learned various lessons that were critical not only for this project but also for future software development projects. First of all, our test coverage is only 75%. This means that while covering the main functionality, our code may still have edge cases or unique scenarios that must be accounted for. We should increase test coverage to ensure the game is resilient to all possible inputs and techniques. Second, we need more user experience. We receive feedback and suggestions from actual players and improve it. We should increase the user experience to improve the game further. However, in the process of development, many places have developed smoothly. TDD ensures good code quality by forcing developers to write test cases before adding new features. Doing so fully guarantees the correctness and reliability of the software and avoids unnecessary bugs. Second, because only one small function is completed at a time, software can be

released more quickly, and end users can test new parts. Moreover, when a program developed using the TDD method fails, it is easy to find the location of the problem without re-running the entire program.

The link for GitHub is: https://github.com/KKrystalLi/PRT582Guess-the-Number-game.git