PRT582 Assignment 2 Report
Hao SUN

Introduction

Our game, the "Number Guessing Game", is a classic challenge where a player's analytical skills are put to the test. The primary objective of the game is to guess a four-digit number randomly generated by the program, with the player receiving hints based on the accuracy of their guesses. As with any software, it's paramount to ensure the reliability and efficiency of our game, and for that purpose, we will be employing the automated unit testing tool - PyUnit.
The game sets a straightforward yet intriguing task for the player. Upon start, a four-digit number is generated in the background, unseen by the player. The game's interactive nature compels the player to keep guessing this number. However, the player is not left entirely in the dark. With each guess, the game provides clues in the form of 'circle' and 'x' symbols. The 'circle' indicates that a digit in the guess is correct and positioned correctly, while an 'x' means the digit is correct but placed incorrectly. After the triumphant moment when the player correctly identifies the number, the game graciously displays the total number of attempts it took. Additionally, players are given the choice to either indulge in another round or gracefully exit the game. One of the most accommodating features is the option for players to quit anytime, ensuring user-friendliness and respecting the player's time and decisions.
To guarantee the game's smooth functioning and validate its requirements, we will utilize PyUnit, a unit testing framework for Python. Through automated tests, we will be able to ensure that the number generation is random, the hinting system works accurately, and the game records and displays attempts correctly, among other functionalities.

Process

Test Case 1: generate_random_number
Our generate_random_number function is designed to generate a random four-digit number and return it as a string. The goal of this function is to ensure that the game has a random yet valid number for the user to guess each time. To guarantee the reliability and correctness of our function, we've outlined the following test scenarios:

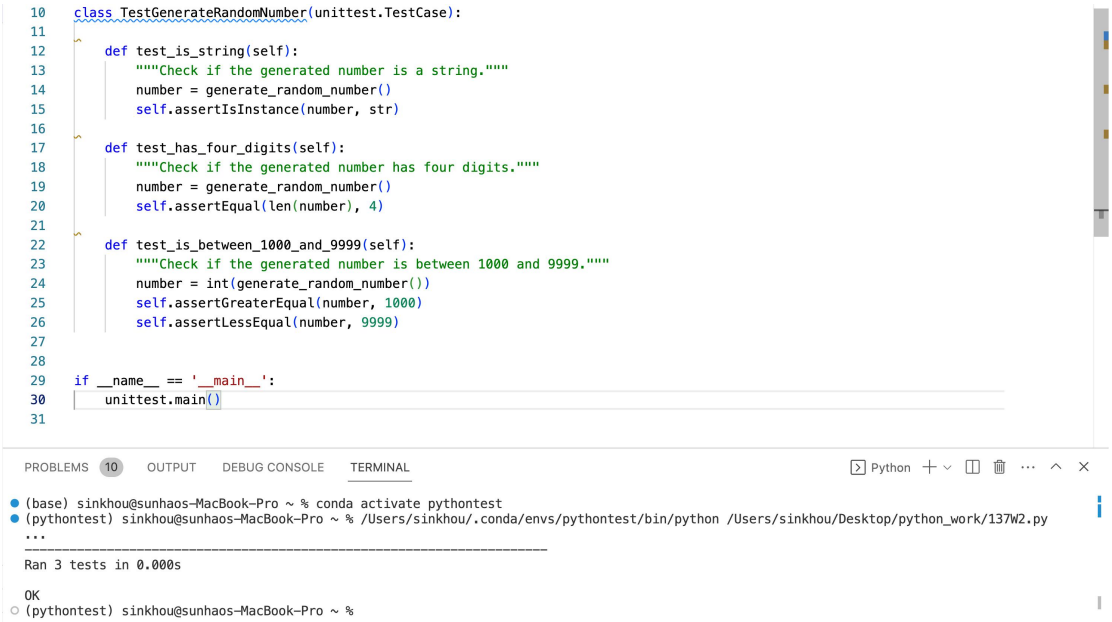
The generated number must be of the type string.
The generated number should consist of exactly four digits.
The number should lie between 1000 and 9999, inclusive.
Test Functions:

test_is_string: This test aims to validate the data type of the generated number. It checks if the output from the generate_random_number function is indeed a string. This ensures that the function's return type is consistent with our expectations.

test_has_four_digits: As evident from its name, this test verifies that the generated number has exactly four digits. This is crucial because a three-digit or five-digit number wouldn't be appropriate for a game that requires a four-digit guess.

test_is_between_1000_and_9999: This test is pivotal in ensuring the bounds of our generated number. We want to make certain that the number is neither too low (below 1000, which would mean it's a three-digit number) nor too high (above 9999). This test confirms that our random number generator is both inclusive of 1000 and 9999 and does not exceed these limits.

By rigorously testing these scenarios, we ensure that our generate_random_number function maintains a standard of reliability and consistency. This, in turn, promises a smooth gaming experience for the user.



```
class TestGenerateRandomNumber(unittest.TestCase):

    def test_is_string(self):
        """Check if the generated number is a string."""
        number = generate_random_number()
        self.assertIsInstance(number, str)

    def test_has_four_digits(self):
        """Check if the generated number has four digits."""
        number = generate_random_number()
        self.assertEqual(len(number), 4)

    def test_is_between_1000_and_9999(self):
        """Check if the generated number is between 1000 and 9999."""
        number = int(generate_random_number())
        self.assertGreaterEqual(number, 1000)
        self.assertLessEqual(number, 9999)


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

```
PROBLEMS 10  OUTPUT  DEBUG CONSOLE  TERMINAL                                  Python

(base) sinkhou@sunhaos-MacBook-Pro ~ % conda activate pythontest
(pythontest) sinkhou@sunhaos-MacBook-Pro ~ % /Users/sinkhou/.conda/envs/pythontest/bin/python /Users/sinkhou/Desktop/python_work/137W2.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
(pythontest) sinkhou@sunhaos-MacBook-Pro ~ %
```

Test Case 2: get_hints
Our get_hints function aims to provide hints for a user's guess in the number guessing game. Given a number and a guess, the function returns a list of hints based on the comparison between the guessed number and the actual number. The possible hints are 'O', 'x', and '-'.

To ensure our function works correctly, we need to test the following:

All digits in the guessed number are correct and are in the right position.
Some digits in the guessed number are correct and are in the right position, while others are not present in the actual number.
Some digits in the guessed number are correct but are in the wrong position.
None of the digits in the guessed number are correct.
Test Functions:

test_all_correct_position: This test checks if the hints returned are all 'O' when all digits in the guessed number match the actual number and are in the correct position.

test_some_correct_position: This test checks the scenario where some digits in the guessed number match the actual number and are in the correct position, while others are not present in the actual number. The expected hints are a mix of 'O' and '-'.

test_some_correct_wrong_position: As the name suggests, this test ensures that when some digits in the guessed number are correct but are in the wrong position, the hints returned are all 'x'.

test_no_correct_digits: This test verifies that when none of the digits in the guessed number match any digit in the actual number, the hints returned are all '-'.

By conducting these tests, we can ensure that our get_hints function is robust and provides accurate feedback to the user based on their guess.

```python
class TestGetHints(unittest.TestCase):

    def test_all_correct_position(self):
        """Test case where all digits are correct and in the right position."""
        number = "1234"
        guess = "1234"
        expected_hints = ['O', 'O', 'O', 'O']
        self.assertEqual(get_hints(guess, number), expected_hints)

    def test_some_correct_position(self):
        """Test case where some digits are correct and in the right position, while others are not present."""
        number = "1234"
        guess = "1256"
        expected_hints = ['O', 'O', '-', '-']
        self.assertEqual(get_hints(guess, number), expected_hints)

    def test_some_correct_wrong_position(self):
        """Test case where some digits are correct but in the wrong position."""
        number = "1234"
        guess = "4321"
        expected_hints = ['x', 'x', 'x', 'x']
        self.assertEqual(get_hints(guess, number), expected_hints)
```

```
PROBLEMS 15  OUTPUT  DEBUG CONSOLE  TERMINAL                                Python

(base) sinkhou@sunhaos-MacBook-Pro ~ % conda activate pythontest
(pythontest) sinkhou@sunhaos-MacBook-Pro ~ % /Users/sinkhou/.conda/envs/pythontest/bin/python /Users/sinkhou/Desktop/python_work/137
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
(pythontest) sinkhou@sunhaos-MacBook-Pro ~ %
```

```python
    def test_no_correct_digits(self):
        """Test case where no digits are correct."""
        number = "1234"
        guess = "5678"
        expected_hints = ['-', '-', '-', '-']
        self.assertEqual(get_hints(guess, number), expected_hints)
```

```
PROBLEMS 15  OUTPUT  DEBUG CONSOLE  TERMINAL                                Python

(base) sinkhou@sunhaos-MacBook-Pro ~ % conda activate pythontest
(pythontest) sinkhou@sunhaos-MacBook-Pro ~ % /Users/sinkhou/.conda/envs/pythontest/bin/python /Users/sinkhou/Desktop/python_work/137
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
(pythontest) sinkhou@sunhaos-MacBook-Pro ~ %
```

Test Case 3: play_game

Testing interactive console applications, especially ones with loops and system quit commands, can be intricate. It's essential to carefully structure the code and tests to ensure compatibility. In our case, while we were able to identify specific behaviors to test, challenges with mocking user input and handling system exits made the testing process more complex than anticipated.

Conclusion

Throughout the development and documentation of the "Number Guessing Game", several invaluable lessons have been gleaned:
Testing is Crucial: The application of PyUnit in testing brought forth the significance of validating every component of our software. Ensuring that each function operates as intended not only guarantees a smoother user experience but also simplifies the debugging process.
Feedback is Vital: The hint system, represented by 'O', 'x', and '-', emphasizes the importance of real-time feedback in interactive applications. By constantly guiding the user, we enhance engagement and user satisfaction.
Simplicity Matters: While the game's premise is straightforward, it reinforces the notion that complex mechanisms aren't always necessary to create an engaging user experience. Often, simplicity paired with clear feedback can yield a captivating product.
In retrospect, this project served as a testament to the power of effective software design, user feedback, and rigorous testing.

Please refer to our GitHub repository: https://github.com/Sinkhou/PRT582.git