Software Unit Testing Report

PRT582

Abodh Gajurel

1. Your task is to write a **Guess the Number game** using **Test Driven Development** in **Python**.

The player needs to guess a four digit randomly generated number. The program will then give some clues about the number. Once the player guesses the number correctly, the program will show the number of attempts taken.

The basic game requirements are:

* + 1. Randomly generate a four-digit number
    2. The program will keep asking the user to guess the number until the player guesses it correctly or has quitted.
    3. When the number is entered, the program will respond with hints using ‘circle’ and ‘x’ to show how accurate the guess was.
       - A ‘circle’ indicates that one digit is correct and is in the right spot
       - A ‘x’ indicates that one digit is correct but in the wrong spot
    4. Once the game is finished,
       - The number of attempts taken will be displayed
       - the player will be asked to quit or to play again
    5. Player can quit the game anytime

You can slightly modify the requirements as long as the basic game requirements remain the same. Check with the lecturer first.

Other requirements for this assignment are:

1. Create a Git directory for your assignment (including word or pdf documents and programming code). Make sure to include the link in your report.
2. Submit the following onto Learnline in a zip folder

* A report of the following format:
  + **Introduction**: Outline the objectives and requirements of the game and automated unit testing tool you will be using.
  + **Process**: You should clearly explain how TDD and automated unit testing tool have been used to create your program. Support it with relevant screenshots for each of the requirements.
  + **Conclusion**: conclude the report with lessons learnt and your GitHub link.
* The python file containing your game

# Introduction

The "Guess the Number" game is a straightforward but entertaining Python program made to test players' estimation abilities. In this game, players attempt to guess a four-digit number that is generated at random. To help players find the right answer, the application offers tips in the shape of the "circle" and "x" symbols. Players can continue guessing through an intuitive interface until they either successfully predict the number or decide to give up. The software keeps track of the player's attempts and gives the player the opportunity to resume the game after it is finished. This application not only amuses players but also exemplifies best practices in coding, testing, and user interaction by using Test Driven Development and automated testing techniques.

Objectives:

There are several objectives of this “Guess the Number” game that it aims to achieve. Some of the are listed below:

* Random Number Generation: The program needs to generate a random four-digit number that the player needs to guess.
* User Interaction: The program needs to include a simple user interface that enables players to participate in the game by entering their predictions.
* Evaluation of the Player's Guess: The program needs to assess the player's guess and offer feedback in the form of clues (such as "circle" and "x") to show how correct the estimate was.
* Tracking of Attempts: The computer must keep track of how many times a player has tried to estimate the right number.
* User Quitting: The player should be able to end the game at any time by typing a particular command, such as "quit."
* Game Loop: The software ought to keep running the game until the player correctly guesses the number or decides to quit.
* Display Results: The application should show the player's total number of attempts once the game is finished (either by guessing correctly or quitting).
* Game Restart: The player should have the opportunity to play the game again if they so desire after it has been completed.
* Handling Valid Input: The application should be able to handle a variety of input situations, such as valid guesses, instructions to quit, and other possible user inputs.
* Automated Testing: To make sure that the program operates as intended in many scenarios, it should be created so that it is simple to test using automated unit testing tools.
* Clean Coding Techniques: The code must follow clean coding techniques, which include meaningful variable names, properly organized functions, and pertinent comments.
* Modular Code Design: The game's functionality can be easily extended or changed if the code is organized in a modular and maintainable manner.

These goals combine to provide a "Guess the Number" game that is practical, user-friendly, and entertaining while adhering to best practices in software development.

Requirements:

The "Guess the Number" game has several requirements that we must meet. Some of them are listed below:

* Randomly generate a four-digit number for players to guess.
* Determine the accuracy of each guess, then give the appropriate hints ("circle" and "x") as necessary.
* Give the players a prompt to enter their predictions.
* To keep the game going until the user guesses the answer right or quits, implement a game loop.
* Keep track of every attempt the player has attempted.
* Give users the option to exit the game by typing a specified command.
* After the game is over, display the number of attempts.

These are some of the basic requirements that the program must meet.

Automated Unit Testing

The practice of using software tools to automatically test individual pieces of code, such as methods or functions, as part of a continuous deployment and delivery process is known as automated unit testing. It is used to detect errors and bugs in code early in the development process when repairing them would be simpler and less expensive all around. For this program too, we have used an automated unit testing tool so that we can detect the errors in the earlier stage of the development.

We can utilize the 'unittest' framework, which is a component of the Python standard library, for this "Guess the Number" game project. Python's "unittest" module offers a framework for creating and executing automated unit tests. It enables us to specify test cases and assertions to make sure our code is correct.

# Processes

Test-first methodology is the foundation of test-driven development, or TDD. The development is driven by tests. In the simplest terms possible, test-driven development is nothing more than frequently occurring or quick cycles of testing, coding, and reworking, with testing coming before coding. These cycles move at such a rapid pace that manual operation is not an option. TDD is a methodology based on computer programming that uses brief, frequent iterations (Goyal, 2016). TDD has the following steps:

* Writing Test cases.
* Red - Test case failure.
* Green - Complete the code and pass the new test case.
* Green – Ensure all old test cases also pass.
* Clean up the code by refactoring it.
* Repeat this cycle.

For this assignment, I have also followed these steps to make sure that it produces reliable, bug-free code and fosters a clear understanding of the code's purpose and behaviour.

Test Driven Development (TDD):

TDD is a software development methodology that stresses developing tests before writing actual code. It goes through a cycle of these three steps: Red, Green, Refactor.

1. Red (Writing a failing Test): To test a new feature, we will first create a test case that describes the expected behaviour. The test ought to fail because we haven't yet put the feature into use.

A white background with black and yellow text

Description automatically generated

Figure 1: Writing a failing test.

1. Green (Write Code to Pass the Test): We will now add the bare minimum of code required to pass the failing test.

A white rectangular object with black text

Description automatically generated

Figure 2: Green Phase

1. Refactor (Improve Code Quality): After a test passes, we can rework our code to make it more readable, efficient, and well-structured. Rerunning all tests is necessary to verify that our modifications haven't harmed the current functionality.

Automated Unit Testing Tools:

We can easily run many tests quickly and consistently thanks to automated unit testing technologies that enable us to automatically carry out our test cases. We can utilize testing frameworks like ‘unittest’ or ‘pytest’ in Python. In the context of the "Guess the Number" game, we would employ them as follows:

Writing Test Cases

To define the intended behaviour of our game, we will first create test cases. We'll develop test cases that account for various scenarios for each need.

Test Case 1: Generating Random Number

We can write a test case for Generating Random Number as:

A screenshot of a computer code

Description automatically generated

Figure 3: Test Case 1.

We are checking the 'generate\_random\_number()' method in this test scenario. The test verifies that the generated number is exactly four digits long. When a number is generated, the'self.assertEqual(len(random\_number), 4)' line determines whether it is precisely 4 length.

Now, we will immediately notice that this test case fails when we run it using the 'unittest' test runner because we haven't yet added the 'generate\_random\_number()' method.

As soon as the 'generate\_random\_number()' function is implemented, we will run the test case once more, and it should succeed since the implemented function produces a four-digit number. This serves as an example of the traditional TDD cycle of creating a failed test, putting the code into practice, and passing the test.

A computer code with black text

Description automatically generated

Figure 4: Implementing the function.

Test Case 2: User Interaction

The 'unittest' framework can be used to create the following test case that will imitate user input and validate the input-capturing functionality:

A screen shot of a computer code

Description automatically generated

Figure 5: Test Case 2.

A computer screen shot of text

Description automatically generated

Figure 6: Implementation of the function.

We are evaluating the 'get\_user\_input()' function in this test case. For the purpose of simulating user input with the desired value (in this case, "1234"), we utilize the decorator ‘unittest.mock.patch’. This test case will pass when run through the 'unittest' test runner, demonstrating that the 'get\_user\_input()' function successfully records user input.

In the ‘patch’ decorator, the'side\_effect' option defines a list of values to return for each call to the patched function (in this case, ‘input()’). The list can be altered to imitate various user inputs for testing various scenarios.

Test Case 3: Guess Evaluation

Using the 'unittest' framework, we can create a test case to confirm the accuracy of the 'check\_guess()' method as follows:

A screen shot of a computer code

Description automatically generated

Figure 7: Test Case 3.

A computer code with text

Description automatically generated with medium confidence

Figure 8: Implementing the function.

We're checking the 'check\_guess()' method in this test scenario. We first give an example secret\_number ('1234') and an example guess ('1243'), and then we compare the output of 'check\_guess(secret\_number, guess)' with the predicted hints ('2 circles, 2 xs'). The 'unittest' test runner would pass this test case when we ran it, demonstrating that the 'check\_guess()' function correctly evaluates guesses and offers hints.

The secret number and the guess are compared in the ‘check\_guess()’ function implementation to count the number of correctly placed digits (circles) and the number of correctly placed digits in the incorrect position (xs). Then, it creates a string with the following format: "X circles, Y xs" to represent the clues.

Test Case 4: Game loop and Quitting

Using the 'unittest' framework, we can create a test case to emulate the game loop and quitting feature as follows:

A screen shot of a computer code

Description automatically generated

Figure 9: Test Case 4.

A computer screen shot of text

Description automatically generated

Figure 10: Implementation of the function.

We are checking the quitting behavior of the 'game\_loop()' function in this test scenario. To replicate the user input "quit," we utilize the "unittest.mock.patch" decorator. The test determines whether the response from calling "game\_loop()" equals "Game Over." The 'unittest' test runner passes this test case when we run it, proving that the game loop and quitting functions perform as expected.

The 'while' loop is used in the construction of the 'game\_loop()' function to repeatedly ask the player for input. When the player types "quit," the loop ends and "Game Over" is returned. This straightforward application illustrates the idea of managing user interaction and exiting the game loop.

Test Case 5: Displaying the Results

Using the 'unittest' framework, we can create a test case to determine whether the application shows results accurately, including the number of attempts and restart options as:

A computer code with text

Description automatically generated with medium confidence

Figure 11: Test Case 5.

A screenshot of a computer code

Description automatically generated

Figure 12: Implementing the function.

We are trying the 'display\_results()' method in this test case. We simulate the ‘print’ function using the 'unittest.mock.patch' decorator so that we may record its output and determine whether it corresponds to the desired output. The 'display\_results()' function should successfully pass this test case when run using the 'unittest' test runner. This will demonstrate that the function accurately displays the number of attempts and prompts for resuming the game.

The 'display\_results()' function implementation shows the player how many times they've tried and asks them whether they want to try again. The test case determines whether the desired output was received when the print function was called.

Test Case 6: Automated Testing and Refactoring

It is a good idea to run every test case simultaneously to make sure that no functionality is broken by new changes. Using the ‘unittest’ framework, we can run all the test cases simultaneously and rewrite the code while guaranteeing that tests pass following each change as follows:



Figure 13: Test Case 6.

A screenshot of a computer code

Description automatically generated

Figure 14: Implementation on game.ipynb

We'll make sure that none of the existing functionality has been broken by running all the test cases at once. We will make sure to run the test cases following each update to ensure that they continue to pass while we modify the code to increase readability and maintainability. This iterative technique enhances the code quality while preserving the accuracy of our software.

# Conclusion

The "Guess the Number" game was developed utilizing Test Driven Development (TDD), which has given me important new insights into how software is made. The project was successful in many ways, but there were also certain places where there was need for improvement. Here is a summary of what worked well, what may be improved, and how these lessons can inform future work:

What Went Well:

* Early Bug Detection.
* Documentation Through Tests.
* Modular Design.
* Refactoring

Different Areas that can be Improved

* User Experience.
* Test Coverage.
* Code Readability.

Measures that will help me to improve.

* Engage actual players to try out the game and provide comments on the user interface. This feedback can be used to guide improvements to the interface, rules, and overall flow of the game.
* Spend time developing a comprehensive set of test cases that cover a variety of eventualities, including erroneous inputs, boundary conditions, and unexpected user behaviours.
* Regular peer code reviews can help us find opportunities to make our code more readable and maintainable.

It's crucial to build on these lessons acquired in subsequent projects to produce software that is not just correctly functioning but also well-structured, simple to maintain, and user-friendly. Unquestionably, incorporating thorough testing, code readability, and user-centered design concepts will result in more effective and significant software development attempts.