SOFTWARE ENGINEERING: PROCESS AND TOOLS

[PRT582]

SOFTWARE UNIT TESTING REPORT

Submitted By

Mahesh Chandra Regmi [SID: S389242]

Submitted To Charles Yeo

 CHARLES DARWIN UNIVERSITY,

FACULTY OF SCIENCE AND TECHNOLOGY,

DANALA | EDUCATION AND COMMUNITY PRECINCT,

AUSTRALIA

3 Sept 2025

Table of Contents

[1. Introduction 1](#_Toc207862127)

[1.1. Project Overview 1](#_Toc207862128)

[1.2. Project Objectives 1](#_Toc207862129)

[1.3. Project Requirements 1](#_Toc207862130)

[1.4. Technical Overview 2](#_Toc207862131)

[1.4.1. Programming Language Justification 2](#_Toc207862132)

[1.4.2. Automated Unit Testing Tool Justification 2](#_Toc207862133)

[2. Process 3](#_Toc207862134)

[2.1. TDD Methodology 3](#_Toc207862135)

[2.1.1. Red Phase Implementation 3](#_Toc207862136)

[2.1.2. Green Phase Implementation 3](#_Toc207862137)

[2.1.3. Refactor Phase Implementation 3](#_Toc207862138)

[2.2. Implementation 3](#_Toc207862139)

[2.2.1. Requirement R1 and R2 3](#_Toc207862140)

[2.2.2. Requirement R3 6](#_Toc207862141)

[2.2.3. Requirement R4 7](#_Toc207862142)

[2.2.4. Requirement R5 8](#_Toc207862143)

[2.2.5. Final Test Output 9](#_Toc207862144)

[2.2.6. Linter Implementation 10](#_Toc207862145)

[3. Conclusion 11](#_Toc207862146)

[3.1. What Went Well 11](#_Toc207862147)

[3.2. Areas of Improvements 11](#_Toc207862148)

[3.3. How Improvements can be implemented 11](#_Toc207862149)

[4. Appendix 12](#_Toc207862150)

[4.1. GitHub Repository 12](#_Toc207862151)

# Introduction

## Project Overview

Hangman is an old school favourite, a word game where the goal is to find the missing word. In this report, I have developed a simple console-based Hangman game using Python Programming language in a test-driven development (TDD) approach. The game has two difficulty levels (Basic and Intermediate) with a timer system, includes input validation and a lives-based mechanism.

## Project Objectives

The goal of this project is to build a word guessing game using Test-Driven Development (TDD) methodology with automated unit testing. The primary objectives for this project are listed below.

* Apply the Red-Green-Refactor cycle throughout the entire development process.
* Write unit tests to cover all the game functionality.
* Build a fully functional word guessing game that meets all the specified requirements.

## Project Requirements

I have developed the following requirements for the Hangman game.

|  |  |  |
| --- | --- | --- |
| **Requirement** | **Description** | **Implementation Status** |
| R1: Two Difficulty Levels | Basic (Single Words) and Intermediate (Phrases) | COMPLETE |
| R2: Valid Dictionary Words | Words/Phrases must be from predefined dictionary | COMPLETE |
| R3: Visual Display | Show underscores for missing letters | COMPLETE |
| R4: 15-Second Timer | Timer with visible countdown per guess | COMPLETE |
| R5: Letter Reveal | Reveal all correctly guessed letters | COMPLETE |
| R6: Lives System | Deduct life for wrong guesses. | COMPLETE |
| R7: Win/Loss System | Game continues until win or lives = 0 | COMPLETE |
| Game Loop | Continue until quit, show answer then. | COMPLETE |

## Technical Overview

### Programming Language Justification

I used Python programming language for this project because of the following reasons.

* Python’s syntax is very easy to learn and understand. It allows me to focus on the game logic rather than complex language constructs such as those in system level languages like Rust, C etc.
* Python provides the ***unittest***module as a part of the standard library. This module is very useful and self-sufficient for running tests as well as writing tests. No external dependencies are required for testing.
* The standard library of python is very broad and covers most of the functionalities we need. It provides libraries like ***random*** for word selection, ***time*** for timer functionality and ***enum*** for state management.

Apart from these benefits, Python is a cross-platform programming language. So, our game will run consistently across Windows, macOS and Linux systems.

### Automated Unit Testing Tool Justification

I selected Python’s ***unittest*** framework as the automated testing tool for this project. This choice was made because of the following reasons.

* ***unittest*** is a built-in framework which doesn’t require installing any extra dependencies and ensures consistent testing environment.
* It has standard testing features such as fixtures, assertions, test discovery and detailed reporting.
* It is widely used in industry for Python development.

This framework enabled creation of 21 test cases achieving 60% test-to-code ratio.

# Process

I used the red-green-refactor cycle throughout the development phase of all the components in this project.

Red Phase: Write failing test  
Green: Write minimal code to pass the test  
Refactor: Improve code quality.

## TDD Methodology

### Red Phase Implementation

In the red phase, every test failed. This ensures the test was testing the desired functionality. Tests are written using the ***unittest*** framework as mentioned earlier, with clear descriptive names following the pattern *test\_<functionality>\_<behaviour>*. Each test only focused on one part of the program. This ensures clarity and isolation of the tests. We utilized mock objects to isolate units that are under tests from external dependencies.

### Green Phase Implementation

In the green phase, I wrote minimal implementation to get the tests go from red to green (failing to passing). No additional functionality was implemented beyond what was the actual need to get the tests passing. Each implementation was verified by running the test suite to ensure there are no regressions.

### Refactor Phase Implementation

In this phase, I refactored the code to make it suitable for the new requirements. It includes naming methods appropriately, extracting functionalities and enhancing readability of the code. Performance optimizations are applied as well in the refactor phase but existing functionality is kept intact.

## Implementation

### Requirement R1 and R2

For the first two requirements, I wrote two simple tests to verify if we can construct a object of the Hangman class first. Then after, I checked if the game level stored in that class matches the level I set. I also added one more test to ensure the target of the game is a single word for basic game and multi-word for intermediate game.

Here is the code screenshot of the test. I saved this test in a file *test\_hangman.py*.

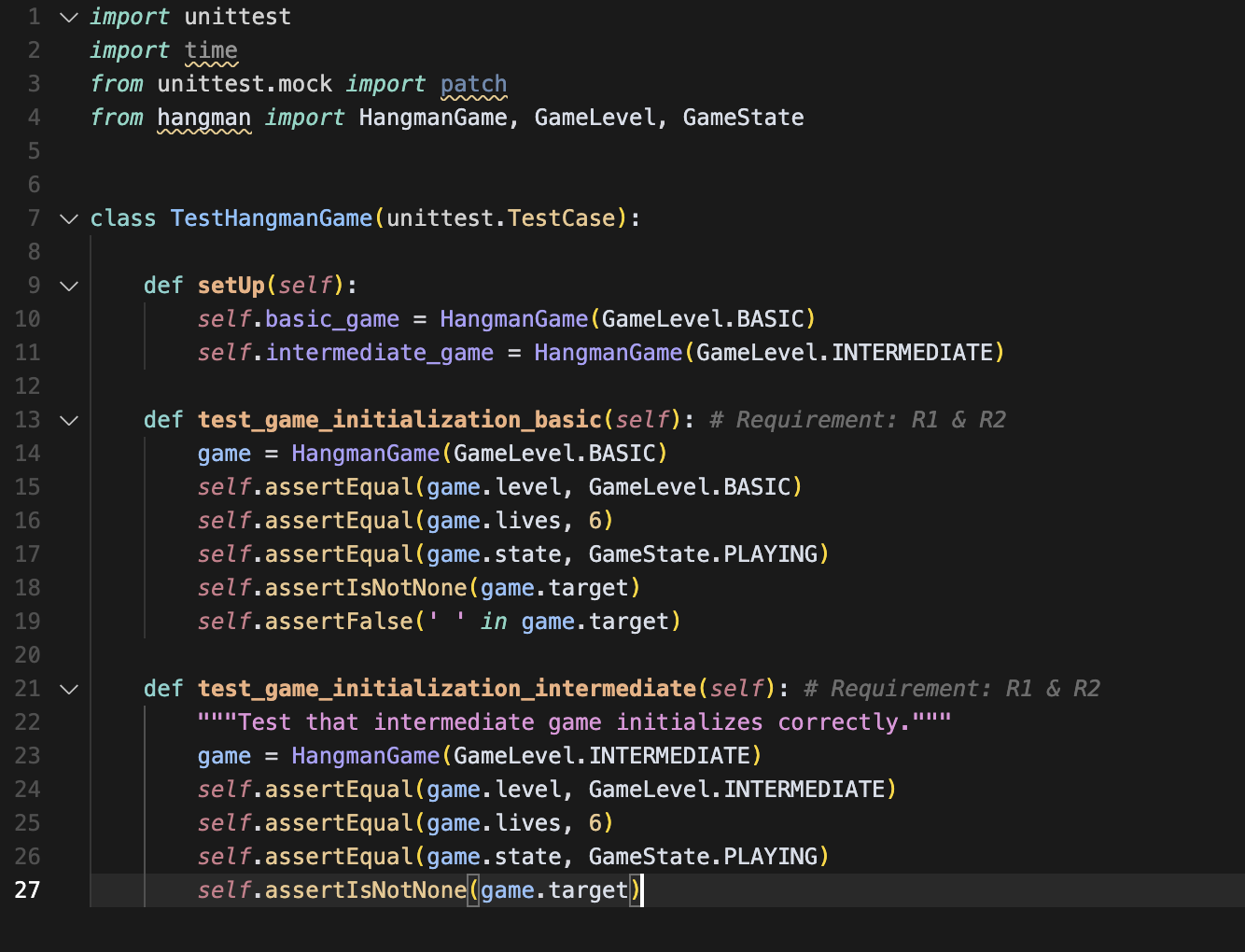


Figure 1: R1 and R2 Tests

These tests failed as expected because hangman module didn’t exist.

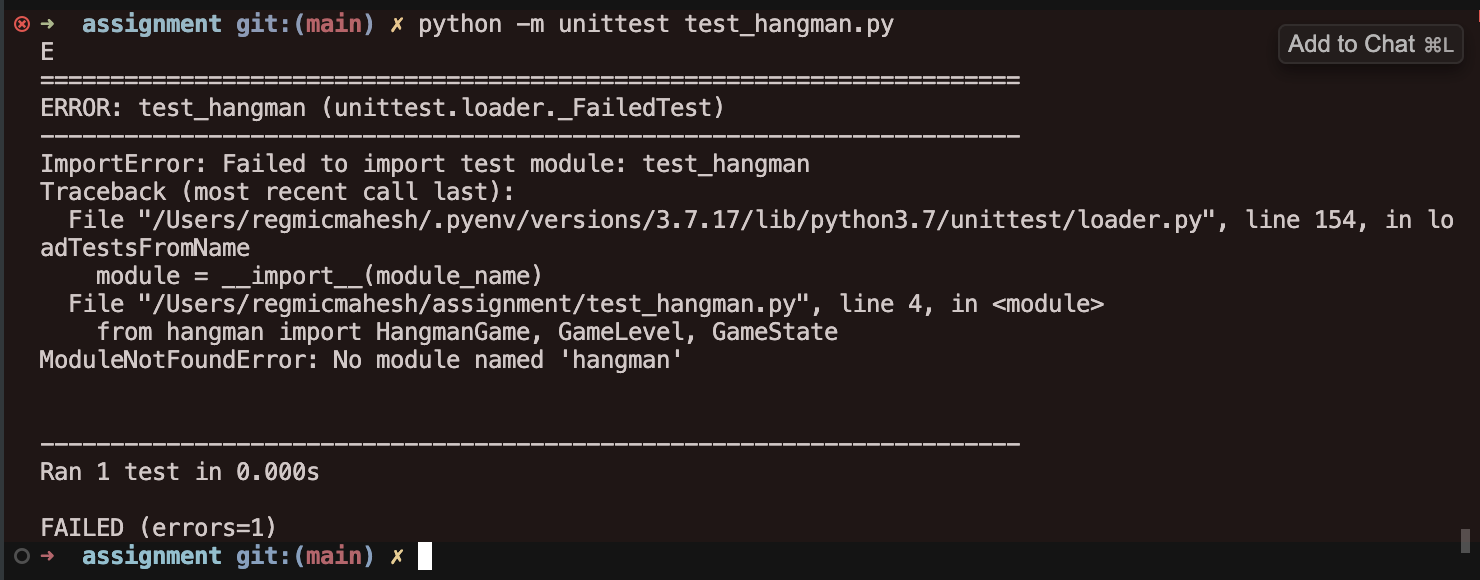


Figure 2: R1 and R2 Tests Failing Screenshot

Once I could confirm these tests are failing, red phase of the TDD is complete. Now, I implemented this behaviour in the file **hangman.py**.



Figure 3: R1 and R2 Implementation

Now, I again re-ran the tests and verified that the tests are running successfully.

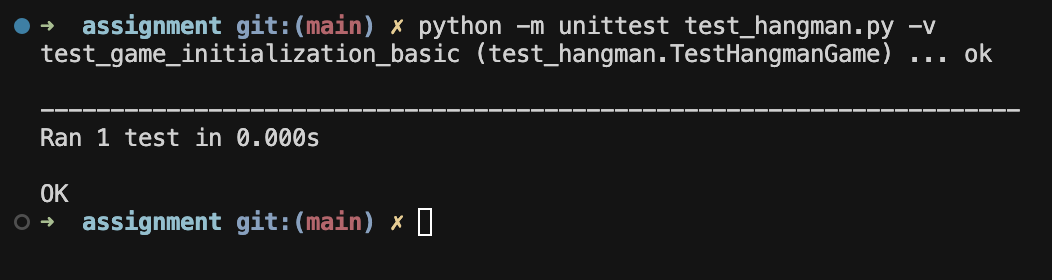


Figure 4: R1 and R2 Test Output

The cycle is complete for the first requirement and now we can move on to the next requirement.

### Requirement R3

I wrote the following tests for the third requirement. I patched the target to **PYTHON** and **HELLO WORLD**. This is required because the word are fetched randomly and we are only checking the display functionality here.

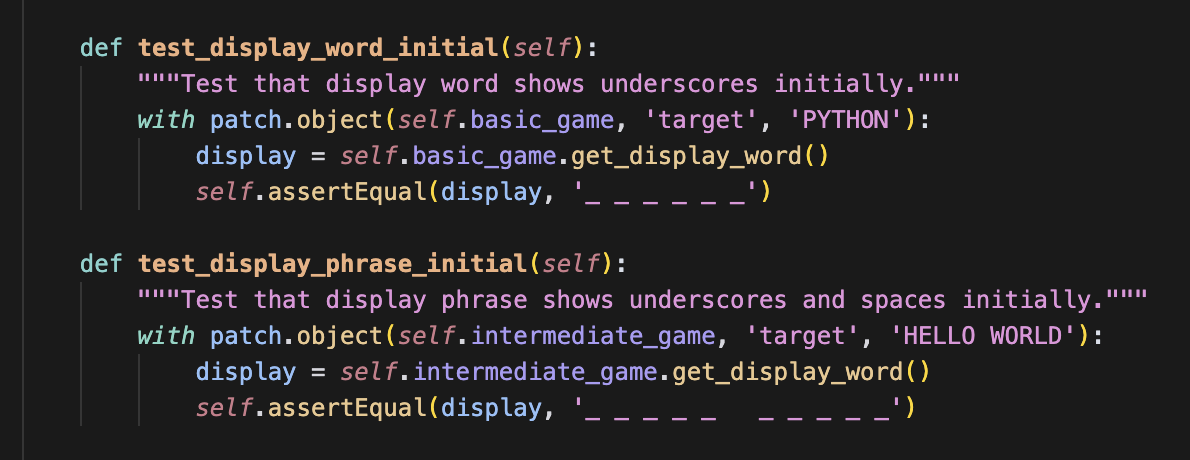


Figure 5: R3 Test

After that, I implemented the functionality to replace all the characters with underscore to get the tests passing. I had to add a new property to the game class called guessed\_letters so that I can display those characters directly instead of underscore for already guessed letters.



Figure 6: R3 Implementation

I refactored the code a little bit and enhanced visual formatting after the implementation in refactor phase for better readability. I followed the similar approach for all other requirements.

### Requirement R4

For the time requirement, I wrote 3 tests to verify the timer functionality. The remaining time should be between 0 and 15 seconds. The timer should not be up before 15 seconds. I also mocked the timer\_start property to expire it and verified that the is\_time\_up works correctly when the time has completed already.

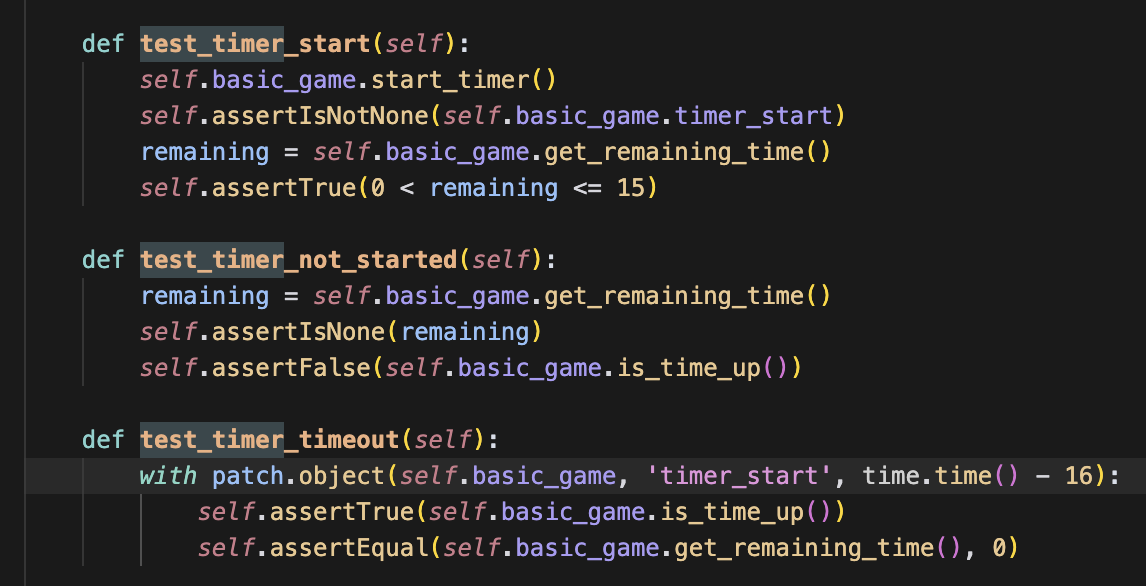


Figure 7: R4 Test

In the green phase of R4, I implemented this functionality with minimal code. The time is first calculated and set to the timer\_start property. It’s lazily evaluated later on to verify it it’s expired already.

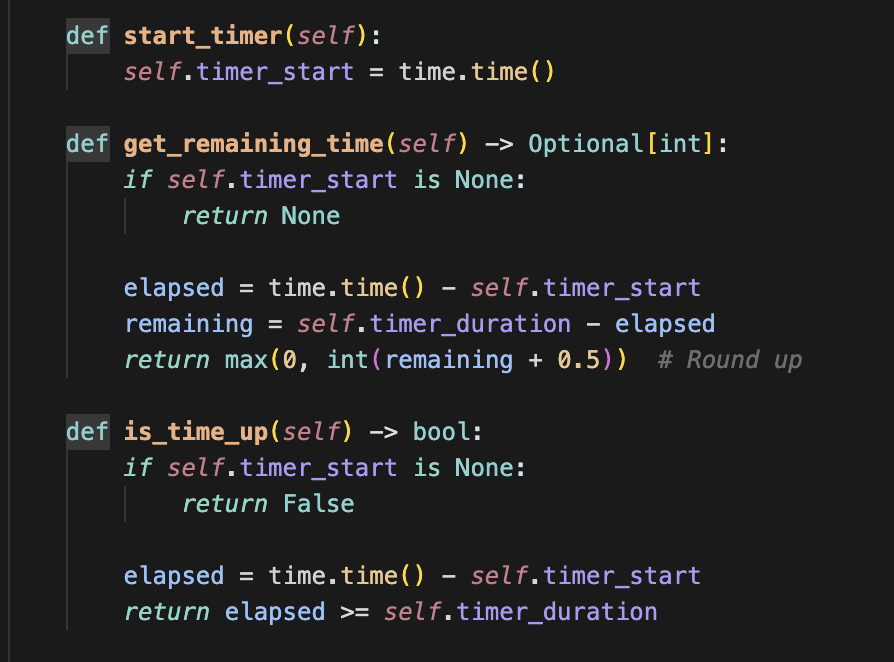


Figure 8: R4 Implementation

### Requirement R5

I wrote a simple test by patching the target to HELLO and checking if the display has accurately displayed the two L characters and other are replaced with underscore.

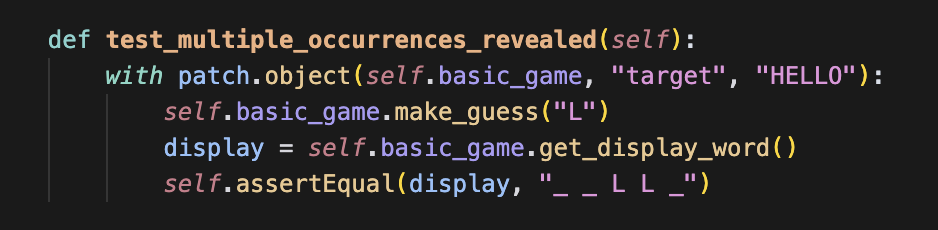


Figure 9: R5 Test

I implemented this functionality in the make\_guess method of the Hangman class. The make\_guess will update the guessed letters with the newly typed letter.

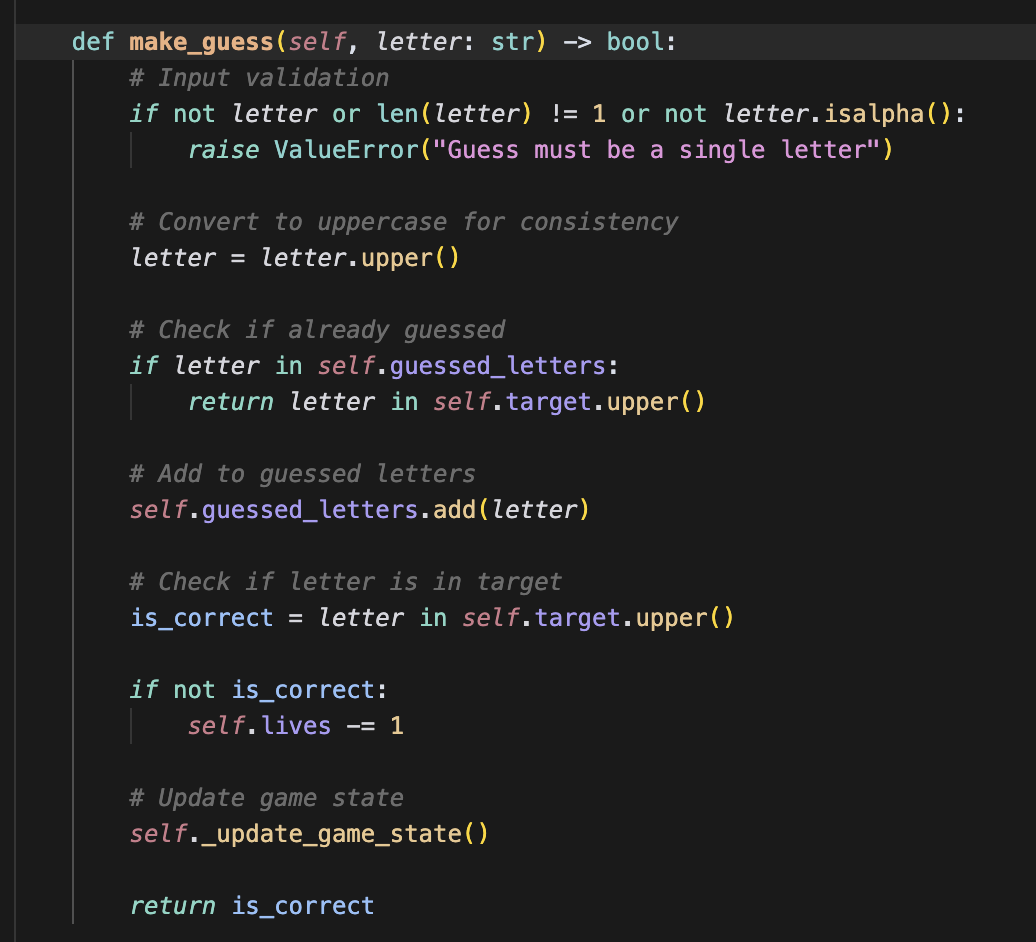


Figure 10: R5 Implementation

Requirement R6-R8 were also implemented in similar way.

### Final Test Output

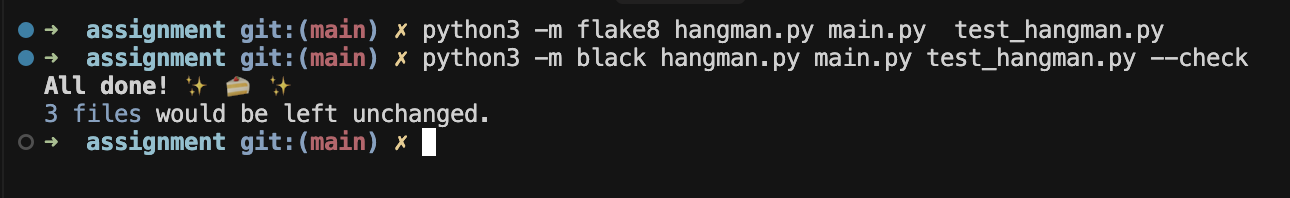
After completing the implementation of the hangman game, I ran all the test suite and we had 21 tests in total. All of the tests were passing which validated all the requirements we had originally listed down.



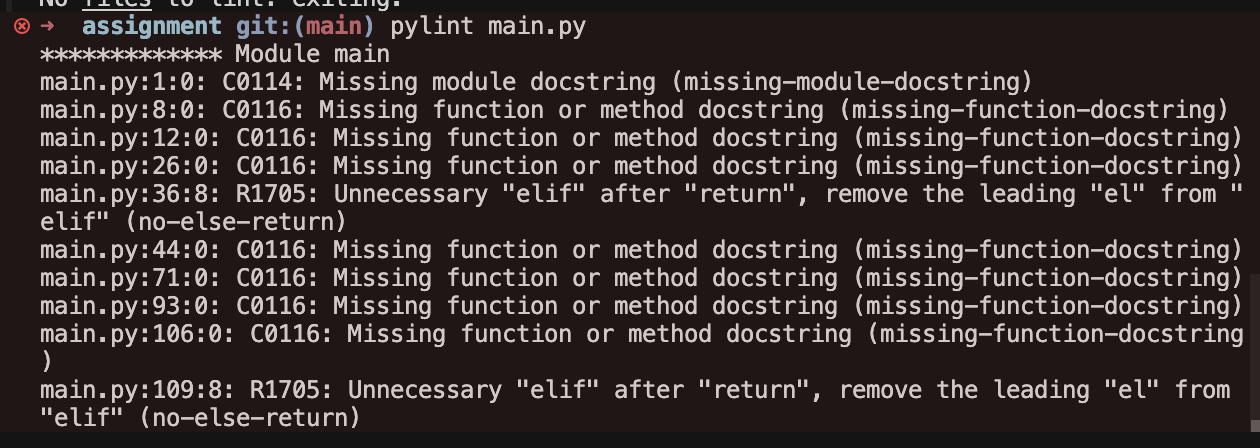
Figure 11: Final Test Output

### Linter Implementation

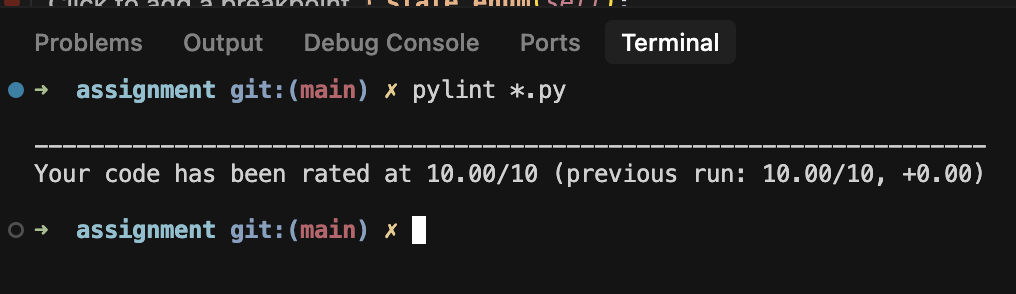
I used ***flake8***, ***black*** and ***pylint*** for linting the code after each implementation.



Originally it had lot of linting errors because of missing comments but I resolved all of those issues by adding docstrings and comments to each and every module, method and class.



There are no linting errors in the program and it satisfied the PEP-8 python standards fully.



# Conclusion

## What Went Well

In this unit testing task, I successfully applied red-green-refactor methodology throughout all the development phases which resulted in well tested and maintainable code.

* The functionality of the program is well covered with 21-unit tests, and it has 100% pass rate.
* All the project requirements are successfully implemented with full functionality.
* There are zero linting errors and the formatting is very consistent and up to the PEP8 standards.
* Multiple quick cycles improved the code quality a lot and provided clear understanding of the system.
* I successfully integrated version control and code quality tools.

## Areas of Improvements

* The timer is lazily evaluated which do not provide a great UX to the user. It could be changed to a more interactive approach.
* The errors messages could be more specific and helpful for different types of invalid inputs.
* Some edge cases were discovered later in the program, and I had to do the work which did not fit in any of the requirement cycle.
* More details in the beginning and more clarity could have made the testing a lot easier.

## How Improvements can be implemented

* For the enhanced timer display, we could use a thread-based approach for real time display. The timer can run on the background thread.
* The error messages can be better handled by creating a separate error handler class and more appropriate error messages.
* Detailed documentation should be developed before and after each approach. This should have corrected the cases where requirements were met but functionality not being correct multiple times.

# Appendix

## GitHub Repository

The repository link is given below.  
<https://github.com/regmicmahesh/tdd-hangman>

It contains the complete source code of the application as well as the development history of the program.