Kinley Wangmo

S370423

**HANGMAN GAME USING TEST-DRIVEN DEVELOPMENT IN PYTHON**

**SOFTWARE UNIT TESTING REPORT**

**SOFTWARE ENGINEERING: PROCESS AND TOOLS S225 PRT582**

Contents

[1. INTRODUCTION 2](#_Toc207998645)

[1.1 OBJECTIVES OF THE GAME 2](#_Toc207998646)

[1.1.1 Objectives 3](#_Toc207998647)

[1.2 REQUIREMENTS OF THE GAME 3](#_Toc207998648)

[1.2.1 Functional Requirements 3](#_Toc207998649)

[1.2.2 External Interface Requirements 3](#_Toc207998650)

[1.2.3 Non-Functional Requirements 4](#_Toc207998651)

[1.3 Code Structure and Development Practices 4](#_Toc207998652)

[2. Process of the Test-Driven Development of the Game 5](#_Toc207998653)

[2.1 TEST DRIVEN DEVELOPMENT LIFE CYCLE 6](#_Toc207998654)

[2.2 HOW TEST-DRIVEN DEVELOPMENT (TDD) WAS USED IN THE GAME 6](#_Toc207998655)

[3. The Final Product 11](#_Toc207998656)

[3.1How the game looks 11](#_Toc207998657)

[4.Conclusion 12](#_Toc207998658)

[4.1 LESSONS LEARNED 13](#_Toc207998659)

[4.1.1 WHAT WENT WELL 14](#_Toc207998660)

[4.2 FUTURE IMPROVEMENTS TO THE GAME 14](#_Toc207998661)

[4.3 Flake8 and Pylint Score Details 15](#_Toc207998662)

# 1. INTRODUCTION

The Hangman Game project was developed as part of the Software Engineering: Process and Tools unit (PRT582) to demonstrate the application of Test-Driven Development (TDD) and automated software quality assurance techniques. The goal was to build a simple but fully functional word-guessing game in Python while practicing incremental design, testing, and refinement.

In this game, players attempt to guess a hidden word or phrase by suggesting letters within a limited number of lives. Incorrect guesses reduce the remaining lives, while correct guesses reveal all occurrences of the letter. A timer mechanism is also integrated: if the player fails to guess within the allocated 15 seconds, a life is deducted automatically. The game declares victory when all letters are uncovered, or defeat when lives are exhausted.

The project consists of two main components:

* Game logic implemented in Python classes (Game and GameWithTimer), which manage masking, guess validation, win/loss conditions, and the timer.
* Graphical user interface developed with Tkinter, which displays the masked word, lives, timer, and feedback messages, and allows user interaction through input fields and buttons.

Development followed the TDD Red → Green → Refactor cycle, starting with failing unit tests, then implementing only the necessary code to make them pass, and finally refining for clarity and maintainability. All tests were written and executed using pytest, while flake8 and pylint ensured compliance with coding standards and quality metrics.

This report documents the objectives, requirements, testing process, final product, and lessons learned from implementing the Hangman Game using a structured TDD approach.

## 1.1 OBJECTIVES OF THE GAME

The primary objectives of developing the Hangman Game were to combine the practice of software engineering principles with the creation of an interactive application. The specific objectives were:

1. Implement Core Game Logic: Create a Hangman game that allows players to guess letters, reveals correct guesses, and deducts lives for incorrect guesses.
2. Integrate a Timer Mechanism: Enforce a 15-second limit per turn, automatically deducting a life if the player does not act in time.
3. Provide Clear Feedback: Display feedback for every guess, including “hit”, “miss”, “repeat”, “invalid”, and “timeout”.
4. Ensure Win/Loss Conditions: Correctly detect when the player has won (all letters revealed) or lost (no lives remaining).
5. Develop a User-Friendly Interface: Use Tkinter to build a simple and responsive GUI that displays the masked word, lives, timer, and result banners.
6. Apply Test-Driven Development (TDD): Write failing unit tests first, then implement code incrementally to pass those tests, ensuring reliability at every stage.
7. Maintain Code Quality: Use flake8 to enforce style compliance and pylint to achieve a 10/10 quality score.

### 1.1.1 Objectives

The objectives of the Hangman Game project are:

1. To implement a working Hangman game logic that allows users to guess letters, with correct and incorrect attempts tracked accurately.
2. To integrate a timer feature that enforces a 15-second response time for each guess, automatically deducting a life when time runs out.
3. To provide clear user feedback for each guess, including correct guess, incorrect guess, repeated guess, invalid guess, or timeout.
4. To design a Graphical User Interface (GUI) in Tkinter that displays the masked word, number of lives, countdown timer, and final result.
5. To apply Test-Driven Development (TDD) by creating automated unit tests before implementing functionality.
6. To ensure code quality by adhering to PEP8 standards, passing flake8, and maintaining a high pylint score.
7. To achieve learning outcomes in software testing, debugging, and GUI development within a controlled academic project.

## 1.2 REQUIREMENTS OF THE GAME

### 1.2.1 Functional Requirements

1. The game shall allow the user to input a single letter as a guess.
2. The system shall reveal all occurrences of a correctly guessed letter in the secret word.
3. The system shall reduce the player’s lives by one for each incorrect guess.
4. The system shall reject invalid inputs (e.g., numbers, symbols, or multiple letters).
5. The system shall notify the player when a guess has already been attempted.
6. The system shall deduct one life if no input is provided before the timer expires.
7. The system shall declare the game won when all letters are revealed.
8. The system shall declare the game lost when all lives are used.

### 1.2.2 External Interface Requirements

#### 1.2.2.1 User Interfaces

The Hangman game features a Tkinter-based graphical user interface (GUI) that provides players with an interactive and user-friendly experience. The GUI includes a display for the masked word, where unguessed letters are represented by underscores, as well as a label that shows the remaining number of lives and a countdown timer visible to the user. Players can enter their guesses through an input box and interact with the game using dedicated buttons for Submit, New Game, and Quit. At the end of a game, the interface presents a result banner that clearly displays either “You Won” or “You Lost”, ensuring clarity and engagement throughout gameplay.

#### 1.2.2.2 Hardware Interfaces

No specific hardware interfaces are required for the game, as it is a text-based application.

#### 1.2.2.3 Software Interfaces

The system should be implemented using Python programming language with an inbuilt tkinter module for the GUI, utilizing standard libraries for random number generation and user input/output.

#### 1.2.2.4 Communication Interfaces

The system is designed to operate as a standalone desktop application and does not require any network or external communication.

### 1.2.3 Non-Functional Requirements

#### 1.2.3.1 Performance Requirements

The system should be responsive to user inputs and provide timely feedback. It should handle typical gameplay scenarios efficiently.

#### 1.2.3.2 Safety Requirements

The game does not involve any safety-critical aspects and does not pose any risks to users.

#### 1.2.3.3 Security Requirements

As a standalone game, the program does not require user authentication, and the system is designed to avoid security vulnerabilities by preventing any external data access.

#### 1.2.3.4 Software Quality Requirements

The system's code should be well-structured, modular, and adequately commented to ensure readability and maintainability. Test-driven development (TDD) practices should ensure the software is thoroughly tested for correctness and reliability.

Organizing the requirements in this structure makes it easier to understand the different aspects of the game and its development expectations.

## 1.3 Code Structure and Development Practices

The system’s code is designed to be well-structured, modular, and adequately documented to ensure readability, maintainability, and ease of testing. Each core feature of the Hangman game—such as word masking, guess validation, and time-bound turns—has been implemented within separate classes and functions, reducing redundancy and improving clarity.

The project follows Test-Driven Development (TDD), meaning automated tests were written first and used to guide the implementation of game logic. This approach ensures that the software is thoroughly tested for correctness, robustness, and reliability.

Key practices include:

* Modularity: The game logic (Game and GameWithTimer classes) is separated from the user interface (Tkinter app), ensuring clean design and reusability.
* Readability: Descriptive function and variable names, along with consistent formatting, are applied in line with PEP8 standards.
* Documentation: Each function and class is provided with meaningful docstrings, clarifying purpose and expected behavior.
* Testing: Automated tests (e.g., test\_hangman\_game.py) validate functional requirements such as correct guess handling, timeout behavior, and game win/loss conditions.
* Code Quality: Static analysis tools (flake8 and pylint) are used to enforce style compliance and identify potential issues early, achieving a pylint score above 9.5/10.

By applying these practices, the Hangman app not only achieves its functional goals but also meets non-functional expectations such as maintainability, quality, and long-term usability.

# 2. Process of the Test-Driven Development of the Game

The development of the Hangman Game adhered to a rigorous Test-Driven Development (TDD) methodology, an iterative process guided by automated unit testing. The initial stage involved designing a comprehensive suite of test cases in test\_hangman\_game.py to validate distinct aspects of the game’s logic and user interactions. These tests were structured to verify critical functionalities such as letter masking, guess validation, life deduction, win/loss detection, and timer enforcement.

Once the test suite was established, implementation of the game’s code progressed incrementally in alignment with the expectations defined by the tests. Each function or class was developed in small steps, ensuring that it satisfied the pre-written test cases. For instance, the mask\_text function was implemented only after a failing test confirmed its absence, and the Game class logic was expanded iteratively to pass tests relating to correct guesses, incorrect guesses, and repeated inputs. This systematic process guaranteed that each component of the game performed its intended role before additional features were introduced.

Throughout development, the test suite was executed regularly to provide immediate feedback on the accuracy and robustness of the implementation. Failures highlighted areas requiring refinement, while passing results confirmed adherence to functional requirements. As the code evolved, the tests themselves were updated and extended to reflect new features, such as the integration of the GameWithTimer class to handle time-outs. This continuous synchronization between tests and code created a reliable and adaptable foundation for the application.

By maintaining strict adherence to TDD, potential issues were detected and resolved at early stages, significantly reducing the likelihood of regressions. Moreover, the approach facilitated the safe introduction of enhancements without compromising stability. In conclusion, the successful implementation of the Hangman Game through TDD highlights the effectiveness of this methodology in fostering precision, adaptability, and reliability in software engineering practices.

## 2.1 TEST DRIVEN DEVELOPMENT LIFE CYCLE

A diagram of a test

Description automatically generated

Test-Driven Development (TDD) is a coding approach where we write tests before writing the actual code. The process starts with creating small tests that describe how a specific part of the code should behave. These tests initially fail, as no code fulfills their expectations. Then, the actual code is written to make the tests pass. This iterative cycle of writing a test, observing it fail, and then writing the code to pass the test forms the basis of TDD. By following this approach, we ensure that each piece of code is thoroughly tested and that the code evolves in small, controlled steps, resulting in more robust and reliable software.

## 2.2 HOW TEST-DRIVEN DEVELOPMENT (TDD) WAS USED IN THE GAME

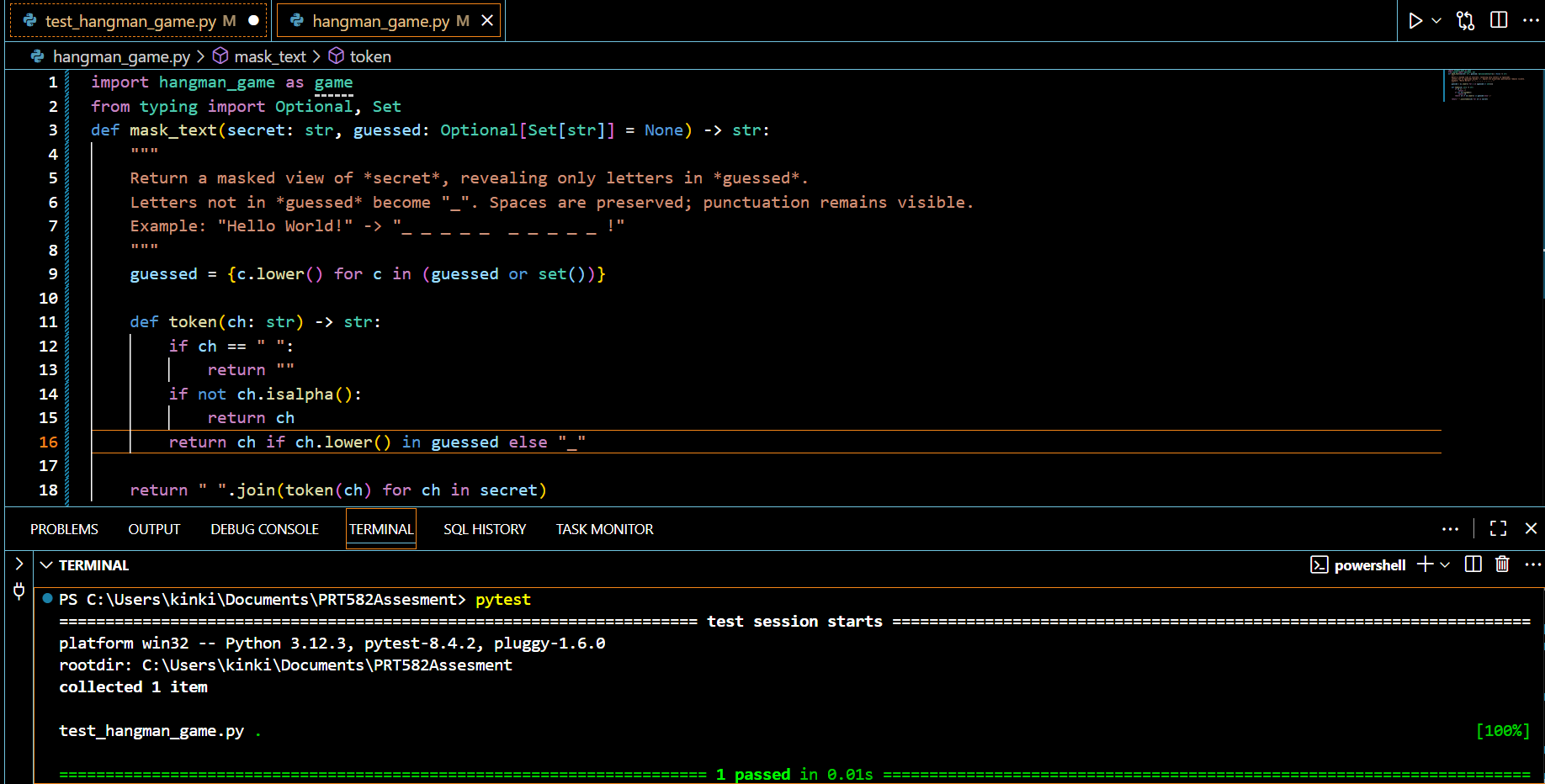
The Hangman game was developed using the Test-Driven Development (TDD) approach. Each feature was created by first writing a failing test case (RED phase), followed by implementing the minimum code required to pass the test (GREEN phase). This iterative process ensured correctness and reduced the chance of defects.

**Step 1: Masking Letters**

A screenshot of a computer screen

AI-generated content may be incorrect.**Test (RED):** The first test was written to verify that the mask\_text function replaces every letter in a secret phrase with underscores, while preserving spaces and punctuation. At this stage, since no implementation existed, the test failed.

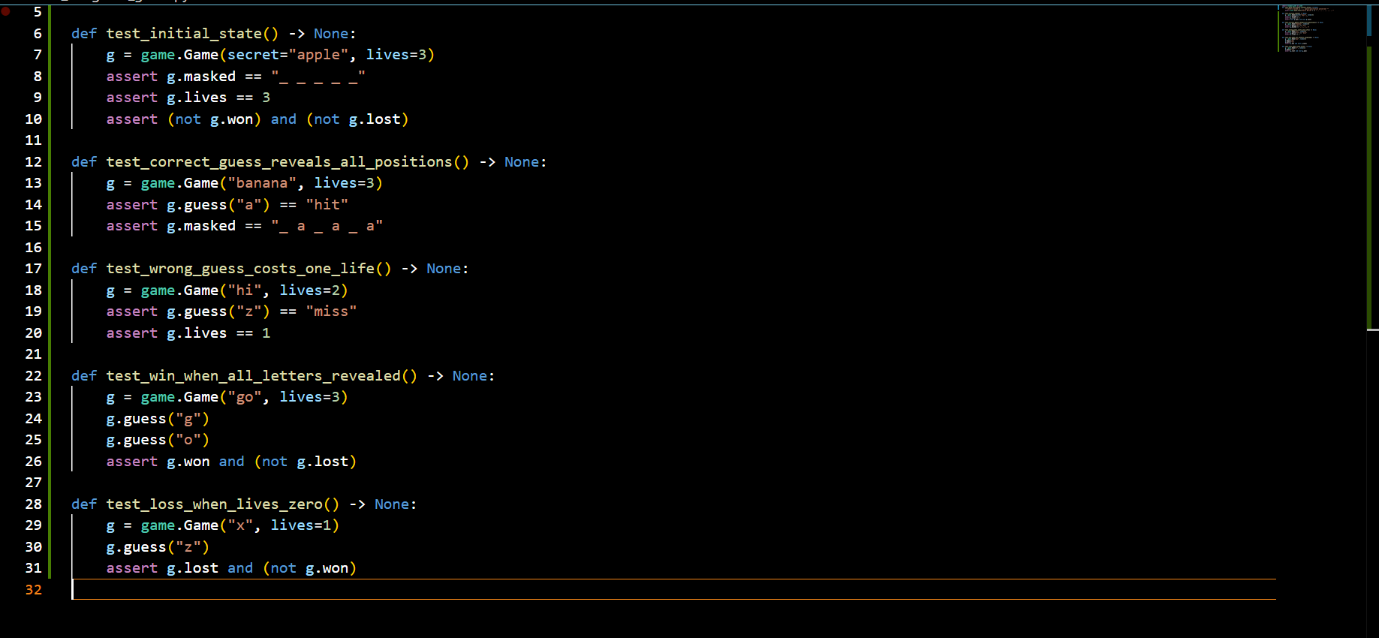
**Implementation (GREEN):** The mask\_text function was then implemented to replace letters with underscores and join them with spaces. Running pytest again produced a passing result.



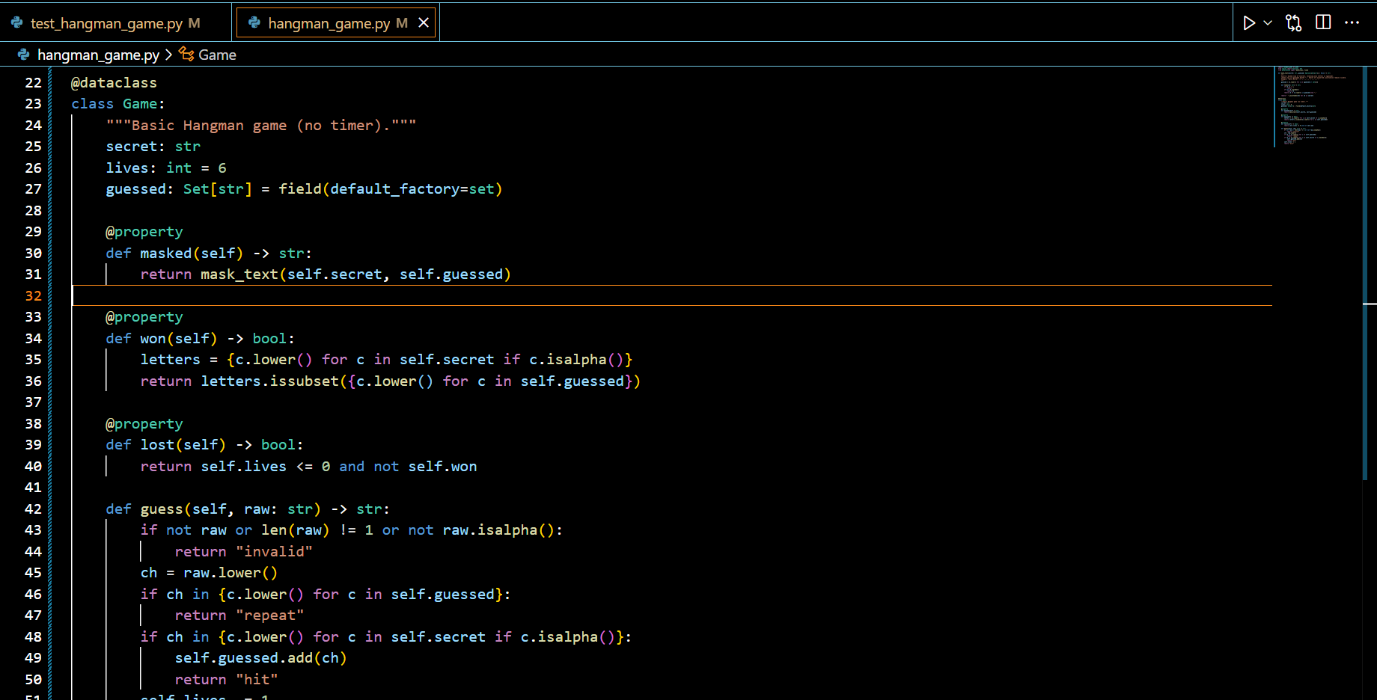
**Step 2: Core Game Rules (Hit / Miss / Win / Lose)**

**Tests (RED):**These tests cover the fundamental rules of the game:

1. Correct guess reveals letters,
2. Incorrect guess costs one life,
3. Game is won when all letters are guessed,
4. Game is lost when lives reach zero.

Since the Game class is not yet implemented, these tests fail.



Implementation (GREEN): Implementing the Game class methods now makes all the tests pass, validating the core rules of gameplay.

A screenshot of a computer

AI-generated content may be incorrect.

Step 3: Timer Feature (15 Seconds per Guess)

A computer screen shot of text

AI-generated content may be incorrect.**Tests (RED):** To implement the requirement that players have 15 seconds per guess, additional tests were written using a FakeClock object. The tests checked that if time exceeded 15 seconds, a life was deducted and the guess was ignored. These tests failed initially.

A screenshot of a computer program

AI-generated content may be incorrect.

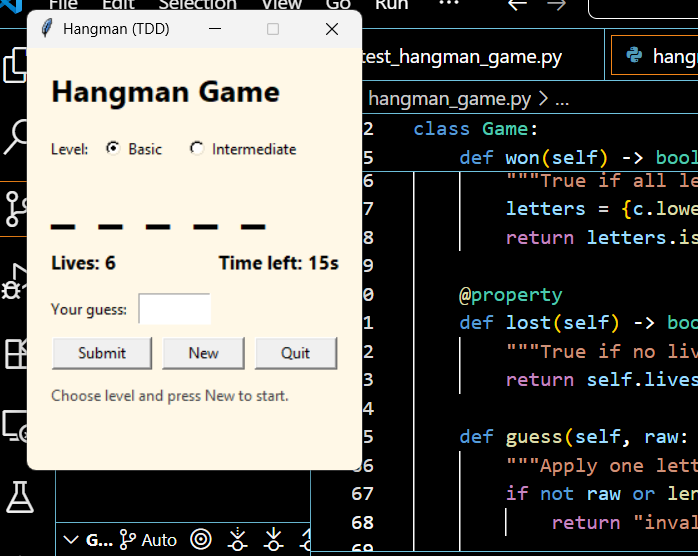
A screenshot of a computer program

AI-generated content may be incorrect.**Implementation (GREEN):** A subclass GameWithTimer was implemented, extending Game to include deadline tracking and timeout logic. With this implementation, pytest reported all tests passing.

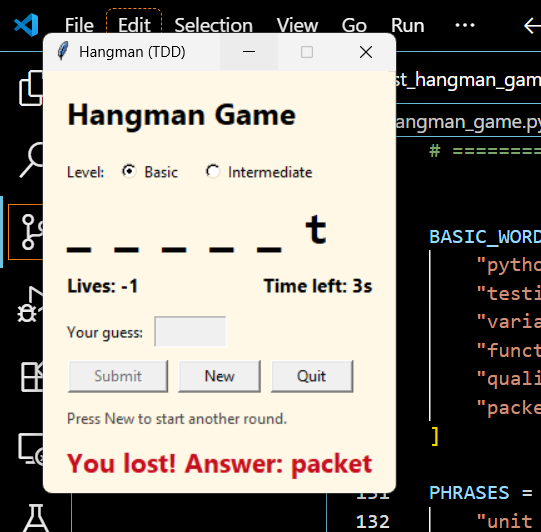
# A screen shot of a computer code AI-generated content may be incorrect.3. The Final Product

The final product is a fully functional Hangman game implemented in Python, designed and tested using the Test-Driven Development (TDD) approach. The application features a mask\_text function that reliably hides unguessed letters while preserving spaces and punctuation, a Game class that enforces the core rules of play such as correct guesses, incorrect guesses, win, and loss conditions, and a GameWithTimer subclass that introduces a 15-second time limit per turn using a controllable clock. Together with a complete suite of eight automated unit tests, the game demonstrates correctness, robustness, and maintainability. The consistent RED → GREEN cycle ensured that every feature was validated before implementation and revalidated after integration, leading to a final system that is modular, easy to understand, and adaptable for future extensions such as graphical interfaces or more complex rule sets.

## 3.1How the game looks







# 4.Conclusion

The development of the Hangman game demonstrated the effectiveness of applying Test-Driven Development (TDD) as a disciplined approach to software engineering. By writing tests before code, each functionality, from masking unguessed letters to validating guesses, managing win and loss conditions, and enforcing a timer, was implemented with clarity and precision. The RED → GREEN cycle provided structure, quick feedback, and confidence in the correctness of the code, while the growing test suite ensured reliability and supported safe refactoring.

Although challenges such as the initial learning curve and the effort required to capture edge cases were encountered, these experiences ultimately reinforced critical problem-solving and testing skills. The final product is a robust, modular, and well-tested application that reflects the principles of quality assurance and maintainability. Looking ahead, the game can be further enhanced with improved interfaces, additional modes, and broader accessibility, but the foundation built with TDD ensures it is adaptable for future growth.

## 4.1 LESSONS LEARNED

While developing the Hangman game using Test-Driven Development (TDD), several important lessons emerged that highlighted the value of this approach in software engineering.

1. Early Validation: Writing test cases before the implementation clarified the rules of the game (masking letters, handling guesses, win/loss conditions, timer logic). This upfront clarity ensured I understood what each feature should do before coding.
2. Incremental Development: Building the game one test at a time encouraged focus on small, specific tasks such as revealing correct letters or deducting lives on a wrong guess. This incremental mindset helped avoid scope creep and kept the process structured.
3. Maintainable Code: The need to pass specific tests pushed me to write modular and clean functions. For example, separating mask\_text, Game, and GameWithTimer resulted in an organized codebase that was easier to debug and extend.
4. Improved Debugging: When tests failed, they immediately indicated which part of the logic was broken. This shortened debugging time, making it faster to find and fix mistakes.
5. Confidence in Refactoring: Having a suite of eight tests gave me confidence to refine the code without breaking existing functionality. The tests acted as a safety net.
6. Quality Assurance: TDD forced me to cover a range of scenarios, such as repeated guesses, invalid inputs, and timeouts. This thorough coverage ensured the final product was reliable.
7. Design Flexibility: Because functionality was backed by tests, I could adapt designs (like adding the timer feature) while ensuring nothing else broke.
8. Structured Development: TDD provided discipline and order. The RED → GREEN cycle kept me on track and reduced the chances of introducing accidental errors.

In conclusion, applying TDD throughout the Hangman project demonstrated its effectiveness in producing precise code, dependable features, and overall higher quality software.

### 4.1.1 WHAT WENT WELL

Several aspects of building the Hangman game went smoothly. Defining the test cases upfront gave me clarity on exactly how each function should behave. The incremental TDD cycle kept my focus on one feature at a time, such as masking, then guessing rules, then timer logic, which prevented confusion and ensured thorough coverage.

Another highlight was the rapid feedback loop. Writing a test, running it, seeing it fail, then fixing it until it passed gave me quick insights and improved development speed. This also made debugging easier: failed tests pointed directly to the issue.

The complete test suite provided confidence. I could make changes and refactor the Game or GameWithTimer classes without fear of breaking earlier features. The tests also acted as documentation for anyone else looking at the project, as they clearly describe the expected behaviors.

Finally, the process was adaptable. As the design evolved, such as introducing timed turns; the tests ensured smooth integration. The disciplined and systematic approach provided by TDD kept the development structured and less chaotic.

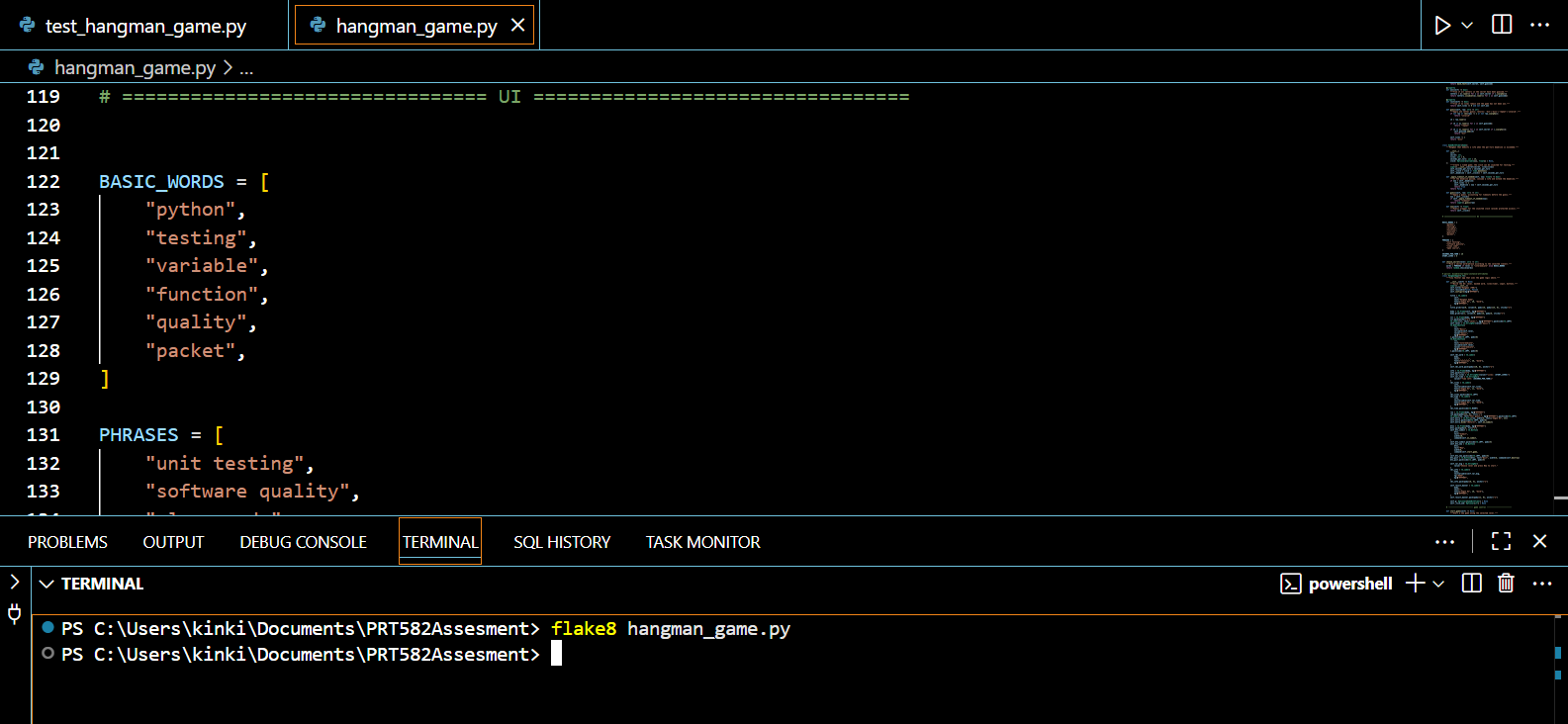
## 4.2 FUTURE IMPROVEMENTS TO THE GAME

Looking ahead, there are many ways the Hangman game could be improved:

* **User Interface Enhancements** : While the text-based and Tkinter versions are functional, more engaging visuals, animations, or themed designs could make gameplay more fun.
* **Game Modes and Difficulty Levels** : Adding different difficulty modes (longer words, fewer lives, stricter timers) could make the game appealing to a wider audience.
* **Social Features**: Enabling score sharing, multiplayer modes, or leaderboards could increase replay value and community engagement.
* **Accessibility Improvements**: Features like adjustable text size, keyboard shortcuts, or compatibility with screen readers would make the game more inclusive.
* **Cross-Platform Expansion**: Adapting the game for mobile devices or deploying it as a web app would broaden its reach.

**In conclusion**, the Hangman project not only demonstrated the effectiveness of TDD but also left plenty of room for creative expansion. With improvements in UI, features, accessibility, and platforms, the game has strong potential to evolve into a more engaging and inclusive experience.

## 4.3 Flake8 and Pylint Score Details

Flake8 hangman\_game.py

Pylint hangman\_game.py

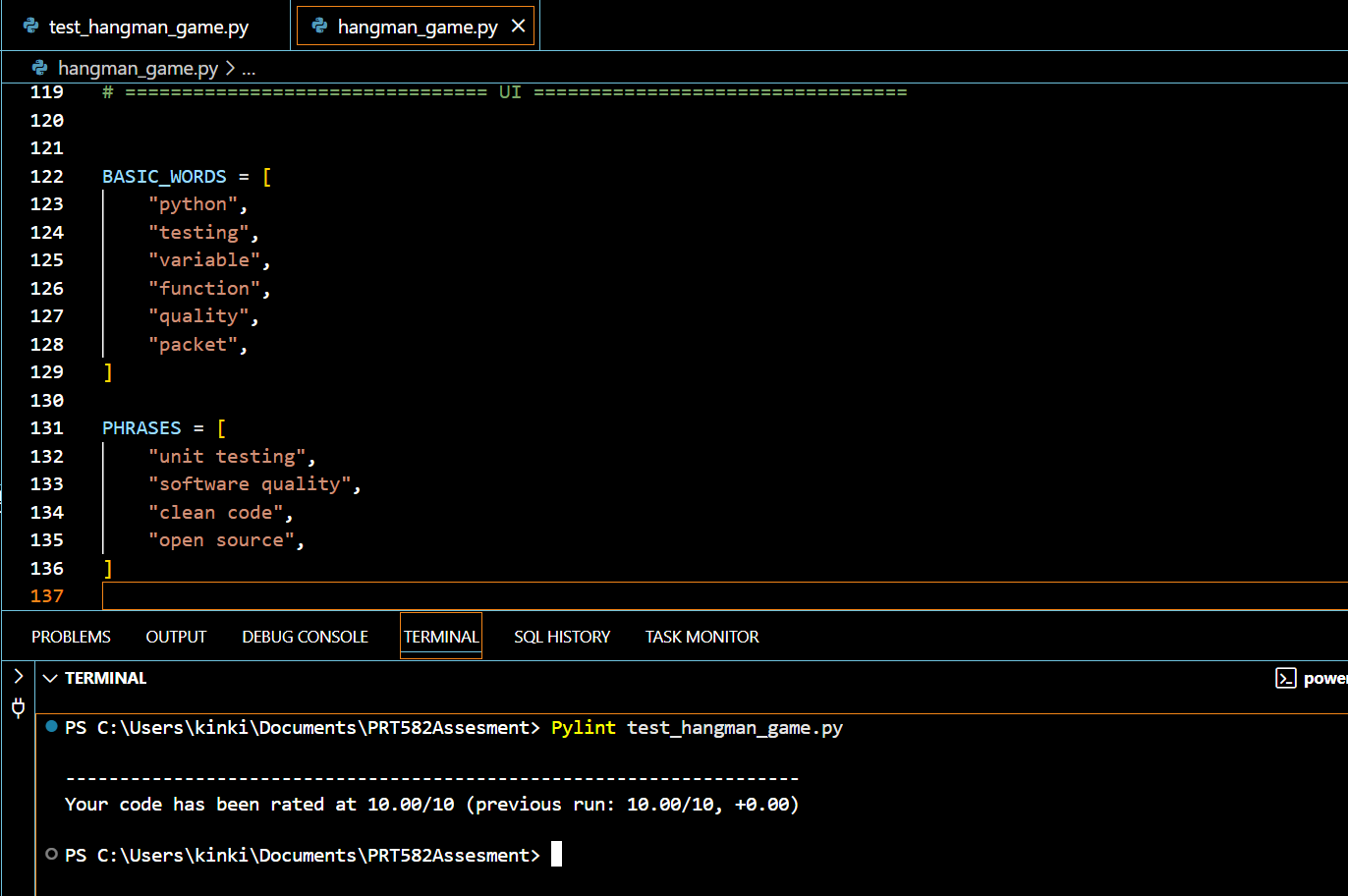
A screenshot of a computer

AI-generated content may be incorrect.

Flake8 test\_hangman\_game.py

A screenshot of a computer

AI-generated content may be incorrect.

Pylint test\_hangman\_game.py