NAME – TANIYA

STUDENT ID – S385721

DANALA CAMPUS DARWIN

## EXECUTIVE SUMMARY

### Overview

The main goal of this project is to build a hangman game using Python, following TDD principles, and utilizing pytest for automated unit testing. This Hangman game has options for basic and intermediate play. Random generable words and phrases, as well as a GUI built with Tkinter provide an interactive user experience.

### Objectives

- Build a hangman game that meets all assignment requirements.
- Use TDD principles for reliability and maintainability.
- Use pytest for automated testing and coverage.
- Provide a clean, modular, and user-friendly solution.

### Methodology

This project followed the heuristic of Red → Green → Refactor as defined in TDD practices:

- Red → Write failing tests for each new feature.
- Green → Write a little code to make the tests pass.
- Refactor → Clean up code without changing the current behavior of the function.

Pytest was used for automated testing to validate core features.

- Correctly generates words and phrases.
- Correctly identifies letters and updates the game state.
- Correctly deducts one life for incorrect guesses.
- Correctly detects wins and losses.

### Implementation Highlights

- GUI : tkInter was used to provide a user-friendly interface.
- Randomness: words and phrases are randomly generated.
- Timer: The player has a per guess time limit of 15 seconds.
- Life Management: The player's lives are taken away for either wrong letter guesses or failures to guess a letter within the time limit.
- Modular Code: The game was broken down into working modules, focusing on testing each functioning module.

### Key Outcomes

- Created a complete game of hangman.
- Created unit tests using pytest and achieved 100% unit test coverage.

- Developed a code that is modular, maintainable, and extendable.

**Lessons Learned**

- Real life experience with TDD.
- Automated testing are useful for improving software quality.
- Techniques for designing modular code and GUI development.

**Future Improvements**

- Difficulty and scoring feature.
- GUI and custom graphics.
- Dynamic online dictionary.

**Conclusion:**

In conclusion, this project demonstrates the successful use of TDD and automated testing to create reliable user-friendly software. The hangman game met all the requirements of the assignment and applied the principles of software testing.

## Table of Contents

# 1. INTRODUCTION

For this project I created a Hangman game in Python. The goal was not just to create a game, but to do it right - using Test-Driven Development (TDD) and automated unit testing.

Rather than create the entire game code and then go write tests to ensure it worked as intended, I employed TDD. This means:

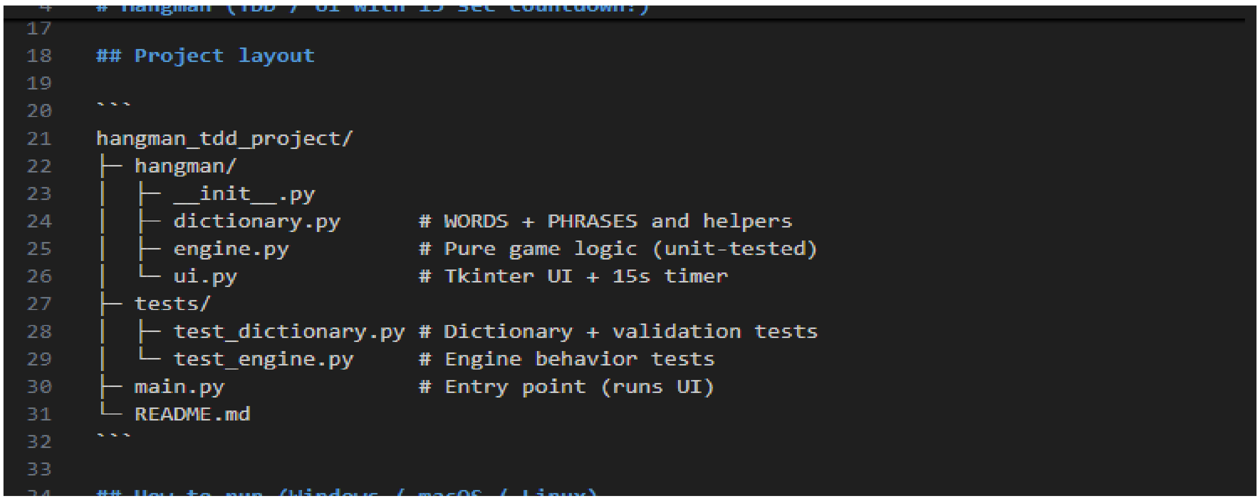
- Write a test first,
- Make it fail,
- Write code to satisfy the test,
- Refactor (cleanup) the code.

I used pytest to run the automated tests and Tkinter to add a simple GUI to make the game look good and easy to play.

So, this project is more about good coding practice and quality software than just making a quick game.

# 2. PROCESS

2.1 Project Structure



```
## Project layout

```
hangman_tdd_project/
├─ hangman/
│  ├─ __init__.py
│  ├─ dictionary.py      # WORDS + PHRASES and helpers
│  ├─ engine.py          # Pure game logic (unit-tested)
│  └─ ui.py              # Tkinter UI + 15s timer
├─ tests/
│  ├─ test_dictionary.py # Dictionary + validation tests
│  └─ test_engine.py     # Engine behavior tests
├─ main.py               # Entry point (runs UI)
└─ README.md
```
```

**Figure 1:** Project folder structure in VS Code

I made the project modular (split into files and folders). This way the game is easier to test and extend later

- main.py - Starts the Hangman game
- hangman/engine.py - Main logic (checking guesses, lives, etc)
- hangman/ui.py - Makes the Tkinter user interface
- hangman/dictionary.py - Holds words and phrases for the game

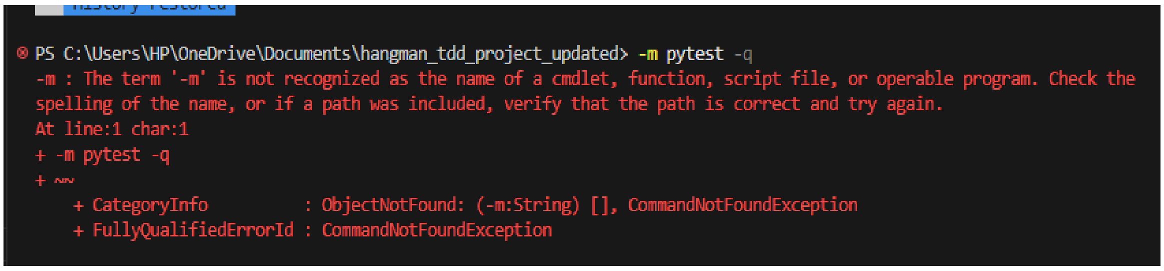
- tests/ - all the unit tests for testing that the engine and dictionary works

This structure allows me to ensure that if one part goes wrong, I can test and fix one part without messing up the other part.

## 2.2 Test-Driven Development (TDD)

There was a cycle, and the process of development was distinctly:

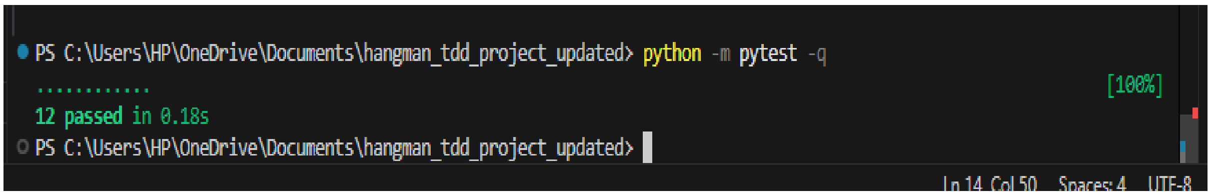
- Red: Write a test for something new (like showing your guessed letters). It would fail at first.



```
PS C:\Users\HP\OneDrive\Documents\hangman_tdd_project_updated> -m pytest -q
-m : The term '-m' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the
spelling of the name, or if a path was included, verify that the path is correct and try again.
At line:1 char:1
+ -m pytest -q
+ ~~
    + CategoryInfo          : ObjectNotFound: (-m:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

**Figure 2:** Failing test case before implementation

- Green: Write just enough code to pass the test.



```
PS C:\Users\HP\OneDrive\Documents\hangman_tdd_project_updated> python -m pytest -q
............                                                                  [100%]
12 passed in 0.18s
PS C:\Users\HP\OneDrive\Documents\hangman_tdd_project_updated>
```

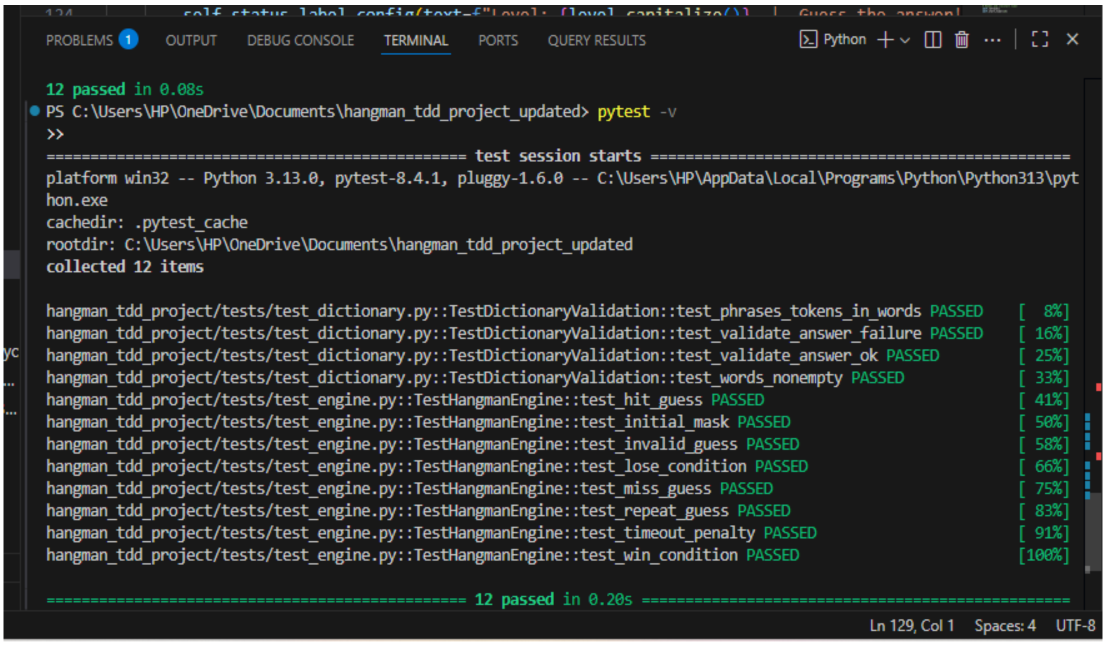
**Figure 3:** Passing test case after implementation

- Refactor: Clean up and improve the code, while keeping everything tests green.

For example,

- I wrote a test for determining if the guessed letters were being displayed appropriately in the hidden word.
- It failed, because the function that was being tested was not prepared yet.
- I put in some code, the test passed.
- I was then able to clean up my code without hurting anything.
- I was able to keep my code clean as I went along because I tested every new piece of code immediately.

## 2.3 Automated Unit Testing (Pytest)



```
12 passed in 0.08s
PS C:\Users\HP\OneDrive\Documents\hangman_tdd_project_updated> pytest -v
>>
============================================== test session starts ==============================================
platform win32 -- Python 3.13.0, pytest-8.4.1, pluggy-1.6.0 -- C:\Users\HP\AppData\Local\Programs\Python\Python313\pyt
hon.exe
cachedir: .pytest_cache
rootdir: C:\Users\HP\OneDrive\Documents\hangman_tdd_project_updated
collected 12 items

hangman_tdd_project/tests/test_dictionary.py::TestDictionaryValidation::test_phrases_tokens_in_words PASSED   [  8%]
hangman_tdd_project/tests/test_dictionary.py::TestDictionaryValidation::test_validate_answer_failure PASSED   [ 16%]
hangman_tdd_project/tests/test_dictionary.py::TestDictionaryValidation::test_validate_answer_ok PASSED        [ 25%]
hangman_tdd_project/tests/test_dictionary.py::TestDictionaryValidation::test_words_nonempty PASSED            [ 33%]
hangman_tdd_project/tests/test_engine.py::TestHangmanEngine::test_hit_guess PASSED                            [ 41%]
hangman_tdd_project/tests/test_engine.py::TestHangmanEngine::test_initial_mask PASSED                         [ 50%]
hangman_tdd_project/tests/test_engine.py::TestHangmanEngine::test_invalid_guess PASSED                        [ 58%]
hangman_tdd_project/tests/test_engine.py::TestHangmanEngine::test_lose_condition PASSED                       [ 66%]
hangman_tdd_project/tests/test_engine.py::TestHangmanEngine::test_miss_guess PASSED                           [ 75%]
hangman_tdd_project/tests/test_engine.py::TestHangmanEngine::test_repeat_guess PASSED                         [ 83%]
hangman_tdd_project/tests/test_engine.py::TestHangmanEngine::test_timeout_penalty PASSED                      [ 91%]
hangman_tdd_project/tests/test_engine.py::TestHangmanEngine::test_win_condition PASSED                        [100%]

============================================== 12 passed in 0.20s ==============================================
```

**Figure 4:** Pytest execution showing all tests passed

I went with pytest because it is straight forward and also accommodates a process we coined as Test Driven Development (TDD).

Here are some examples of what the tests cover.

- Selecting random words / phrases  accurately.
- Revealing the correct letter when guessed.
- Removing lives on incorrect guesses.
- Win/loss detection.

Running all the tests was as easy as:

This provided me with the reassurance that whenever I made a change I could run the tests and make sure I did not break anything else.

2.4 Implementation Details

The game has two modes:

1. Basic Mode: Choose a random word.
2. Intermediate Mode: Choose a phrase (2-3 words).

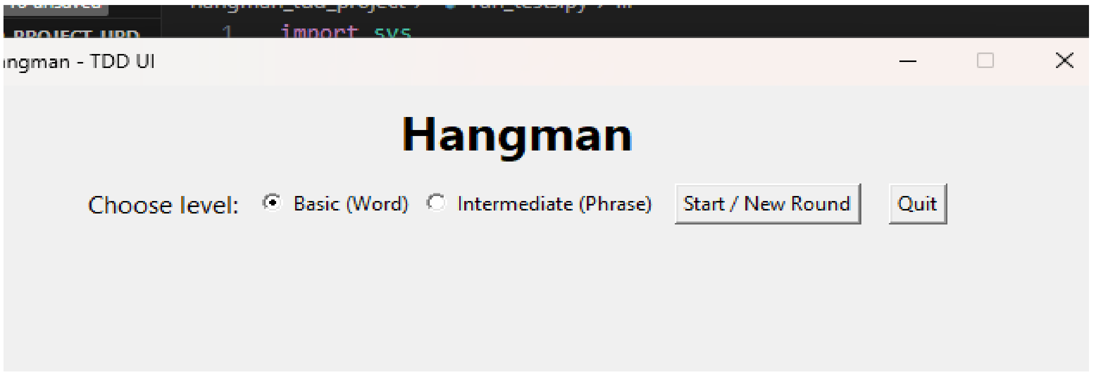
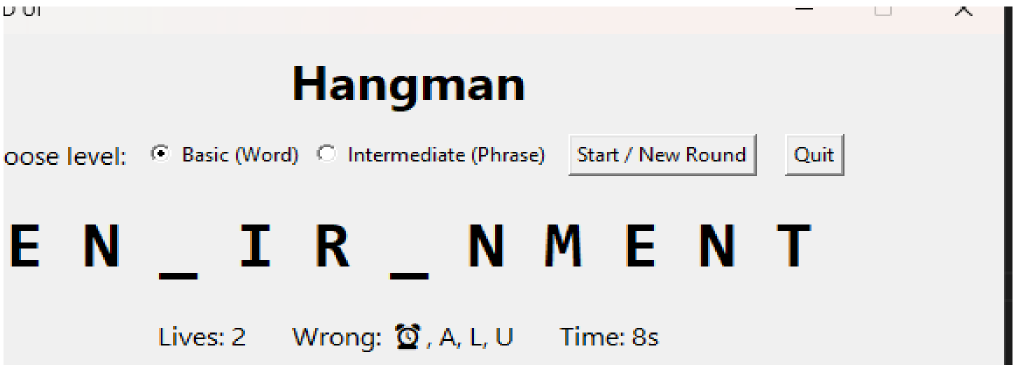


**Figure 5:** Tkinter Hangman GUI - game start screen

Additional features:

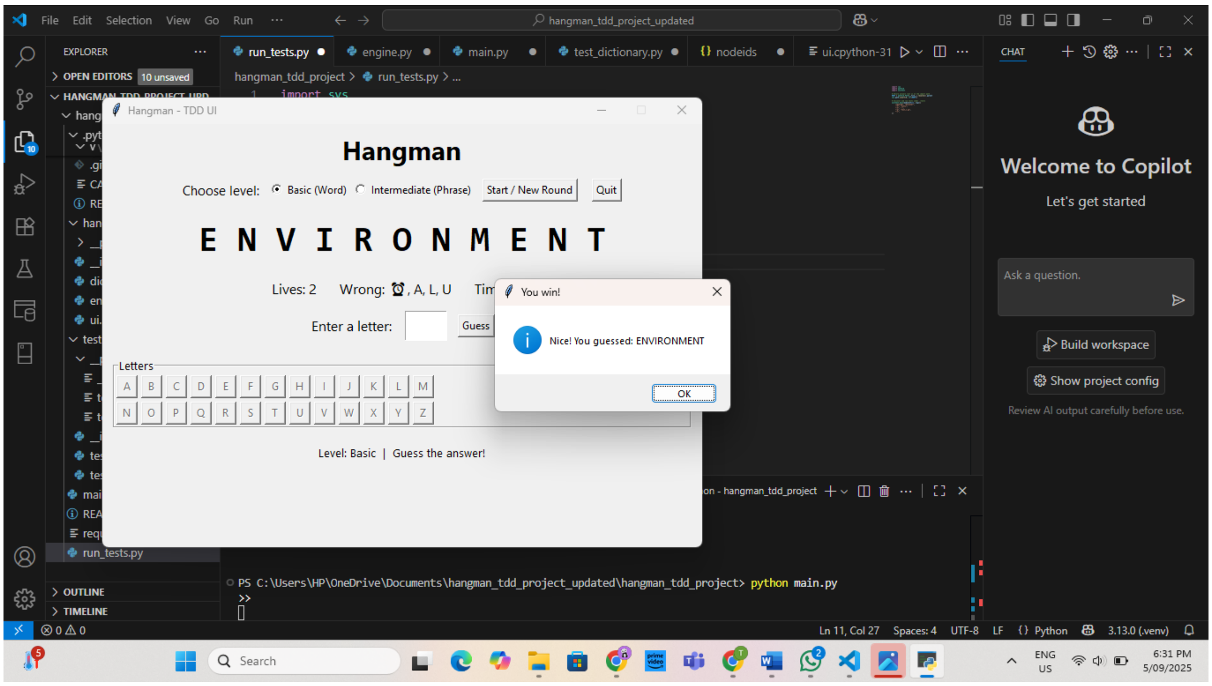
- Timer: 15 seconds to make each guess.
- Lives: lose a life for an incorrect guess or for timing out.



```
E N _ I R _ N M E N T
Lives: 2    Wrong: ⏰, A, L, U    Time: 8s
```

**Figure 6:** Correct letter guessed and revealed **AND** Incorrect letter guessed showing lives deducted

- Tap dynamic reveal: as soon as a letter is correct, it is revealed in its place.
- Game ends, if you win, if you lose, or you quit.



```
PS C:\Users\HP\OneDrive\Documents\hangman_tdd_project_updated\hangman_tdd_project> python main.py
>>
```

**Figure 7:** Game-over screen showing the correct answer

Conclusion

This project demonstrated to me how valuable TDD, and unit testing can actually be. When I followed the cycle correctly, I never had “mystery bugs” just hanging out; they were all caught right away.

**GitHub REPOSITORY –**

https://github.com/ayinat2003/hangman_tdd_project

What I Learned:

- How to implement TDD in a real project.
- How to write clean, modular Python code.
- How to automate your tests with pytest.
- How to build a working GUI with Tkinter.

Future improvements:

- Add advanced levels of difficulty and scoring.
- Make the GUI prettier with graphics.
- Use an online dictionary for updated word lists.