# Introduction

This report is about a program that was written for Hangman game using TDD and automated unit testing tools in python. The game designed in this activity is an old school favourite game. It is a simple word game in which the goal is to guess the missing word. The game has two levels basic and intermediate. In basic mode, randomly a word is generated, and user guesses the word. In the intermediate mode, a phrase is generated randomly, and user guesses the phrase. The user enters the missing letters with 15 seconds of time to enter each letter. A wrong guess deducts a life, and the word must complete before all lives are deducted resulting in no life left. The game goes on until the life is zero and display answer.

## Requirements and Objectives

The project’s objective was to develop and implement the hangman game for guessing word/phrases using Test Driven Development (TDD) that is support by the automated unit testing. The requirements of the game are as follows:

1. Two difficulty levels, basic for word guessing and intermediate level for phrase guessing.
2. The generated word or phrase should be a valid dictionary word/phrase.
3. The underscores are shown to depict the missing letters in word/phrase to be guessed.
4. A guess must be made within given 15 seconds. A life is deducted if timer is out.
5. If guess is correct for a letter, the positions where the letter exists are revealed.
6. If the guess is wrong, a life is deducted from the lives.
7. The guess must be done before all lives are deducted to zero.
8. The game continues to run until
   1. The lives reach to zero, or
   2. Player quits the game, or
   3. The player guesses correctly and doesn’t select to enter the next round.

## Automated Unit Testing Tool

The automated testing tool used in this program was “unittest” framework built-in python. This framework has following functions:

* It has a structured way to define the test cases for verifying each class and function in a program.
* It can be integrated into almost all IDEs and CI/CD pipelines.
* It is the most reliable, suitable and lightweight framework for TDD workflows.

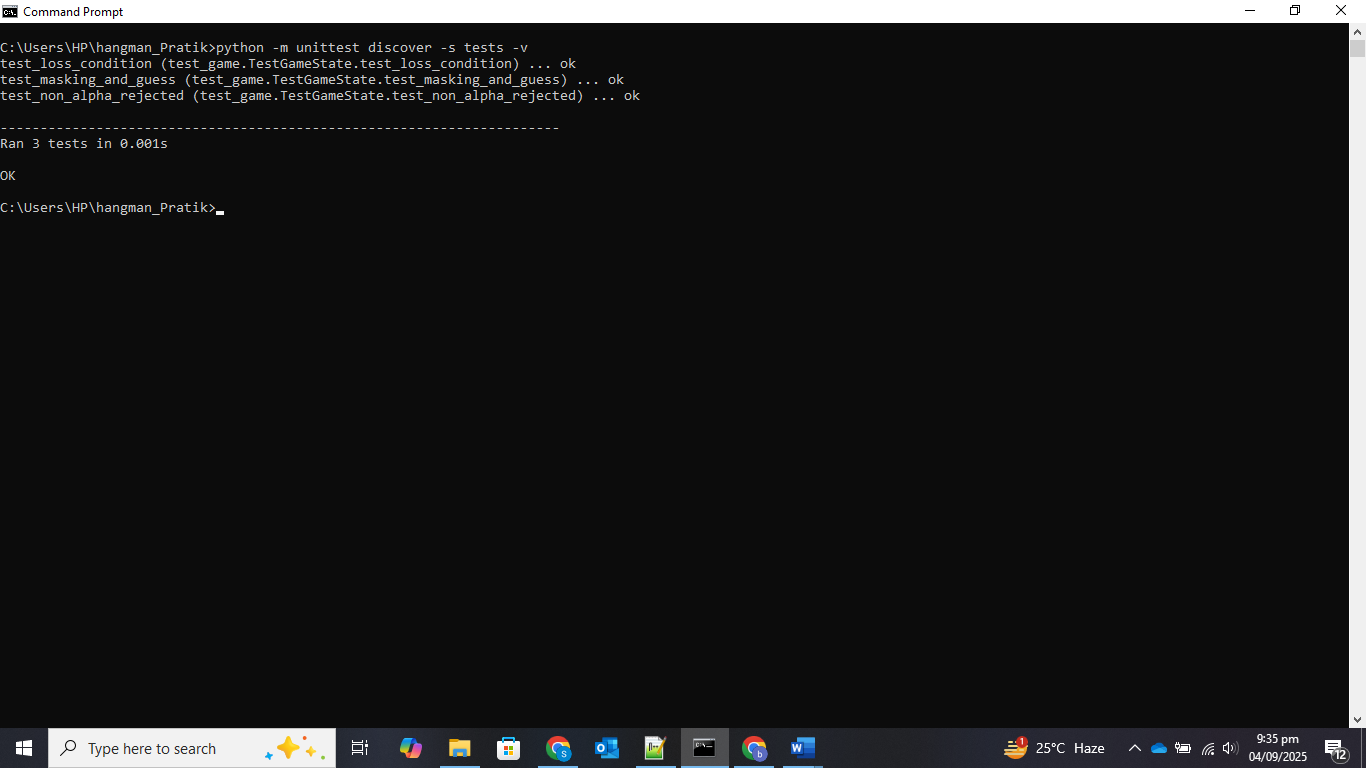


Figure : Unit Testing Results (All Tests were passed)

As we can see in the screenshot given above all the tests were passed. It ran 3 tests for checking the loss conditions, masking and guess as well as the non alpha rejected states.

# Process

## Test Driven Development (TDD)

The TDD cycle was used to implement the project step by step:

1. Writing a failing test for given requirement.
2. Implementing minimum code for making the test pass.
3. Refactoring the code while keeping the tests ok.

For instance, a test was written first for verification that a word can be masked like “Cup” to “---“. The “GameState” class was used to pass the test. Other additional tests were included to test win/loss detection, case sensitivity, repeated guesses and invalid inputs. This ensured that each step was done correctly and debugging time was reduced drastically.

## Automated Unit Testing

All the major requirements of the program were covered using the “unittest” framework in python.

* Work masking was used to check if the underscores and revealed letters are properly and correctly displayed.
* Lives deduction was tested for timeouts and wrong guesses.
* Case insensitivity was also tested to ensure that the small and capital letters are treated in the same way.
* The game termination conditions like loss scenarios and win scenarios were tested.

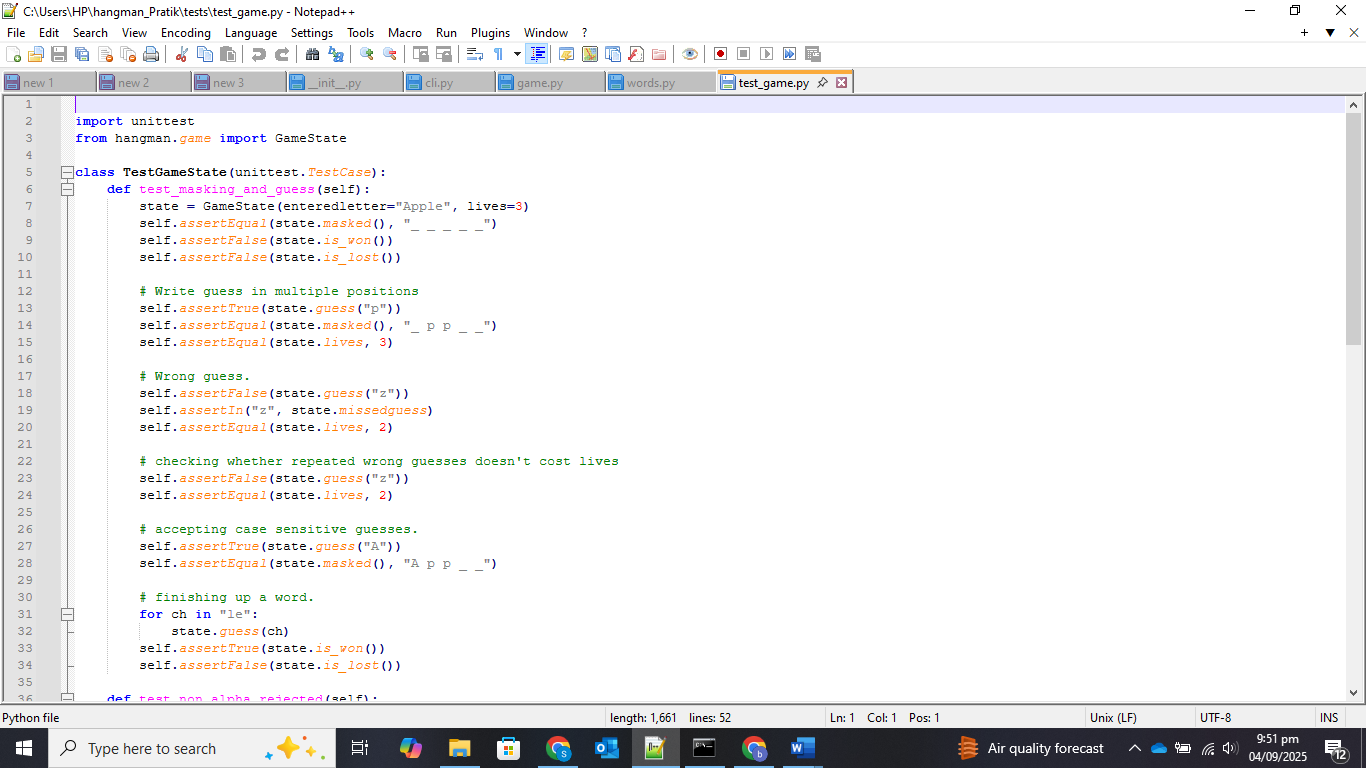


Figure : Code Snippet from test\_game.py

A screenshot of a computer

AI-generated content may be incorrect.

Figure : Masked Word Updates after guesses in Console

## Meeting the Requirements

The requirement to include two modes for the game was met using curated lists in words.py. These two modes included basic/intermediate for random word and phrase generation.



Figure : Game Start Screen

The clear display using dashes “\_” for missing words were done via masked display. This code was implemented in game.py.

A screenshot of a computer

AI-generated content may be incorrect.

Figure : Displaying dashes and letters properly

The timer for 15 second limit was implemented in cli.py that was displayed for each prompt.

A screenshot of a computer

AI-generated content may be incorrect.

Figure : Console showing timer

The life deduction for wrong guesses and timeouts were tested using the testing techniques. The output is shown below

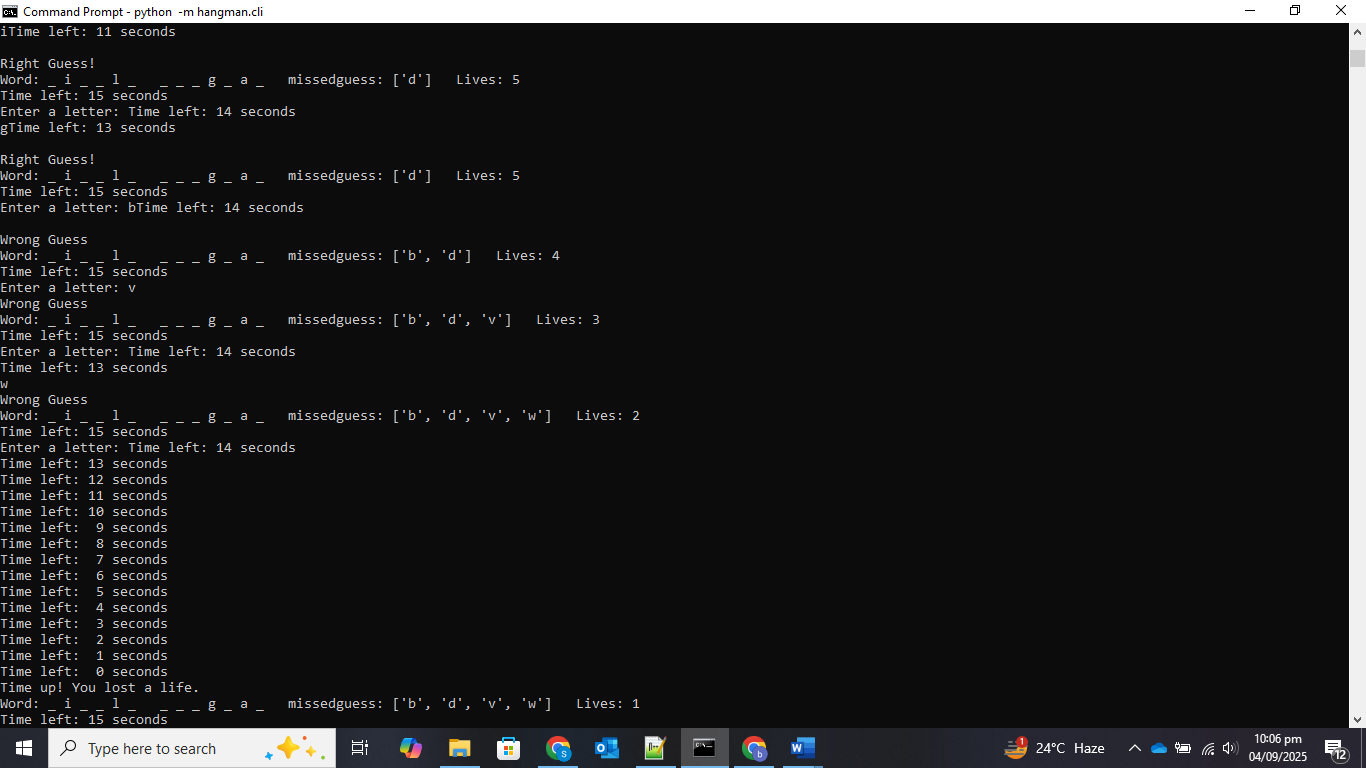


Figure : Lives decreasing for wrong guesses and timeouts

The code ensured that the game continues properly until the player quit, lose or win and terminates the game. This condition was also met successfully.

A screenshot of a computer

AI-generated content may be incorrect.

Figure : Game Continuation and Termination

# Conclusion

The development was structured using TDD where each feature was backed by the test. It enabled catching the bugs in timer and masking were detected earlier and fixed systematically. The UI/logic was built with confidence due to automated tests without any break in functionality.

Initially, the UI suffered due to unclear underscores and overwritten entries onto prompts. The output timer was initially too verbose with spammed lines making the screen scroll unlimitedly. This design has limited words/phrases source dictionary with limited entries that may be replaced by actual dictionary in future.

A more smother and coloured UI may be made using libraries like curses or rich. A larger actual dictionary may be used instead of limited dictionary. ASCII drawings may be used for better feedback with decrease in lives. The CLI interactions may be covered using extended tests. The project is a demonstration of how TDD and automated unit testing can be used for better maintainability and functionality of software.