PRT 582: SOFTWARE ENGINEERING - PROCESS AND TOOLS

ASSIGNMENT – 2

(Software Unit Testing Report)

NAME: Aashish

STUDENT ID: S385593

Table of Contents

Hangman Game Development Report

Introduction

This project is a console-based Hangman game developed using Test-Driven Development (TDD). It features two difficulty levels, a 15-second timer per guess, and a system with 6 lives. Python was chosen for its simplicity and strong testing support. The development followed the TDD cycle of writing tests first, then coding and refactoring. The game passed all 19 unit tests with perfect code quality scores. It has a clean interface, proper error handling, and meets all project requirements. This project demonstrates effective use of TDD to build reliable, well-tested software.

Project Objectives

This report outlines the development of a console-based Hangman game using Test-Driven Development (TDD) methodology. The primary objectives were to:

1. Create a fully functional Hangman game with two difficulty levels (Basic and Intermediate)
2. Implement a timer system with 15-second countdown per guess
3. Develop a life management system starting with 6 lives
4. Apply Test-Driven Development principles throughout the development process
5. Achieve perfect code quality standards using automated linting tools
6. Ensure comprehensive test coverage for all major functionality

Programming Language Justification

Python was selected as the programming language for this project due to several key advantages:

- **Readability and Maintainability:** Python's clean syntax promotes readable code, essential for TDD where code is frequently refactored
- **Rich Standard Library:** Built-in modules like unittest, threading, and random provide all necessary functionality without external dependencies
- **Rapid Development:** Python's interpreted nature allows for quick iteration cycles, crucial for TDD's red-green-refactor approach
- **Strong Testing Ecosystem:** Excellent built-in testing frameworks and tools support comprehensive unit testing
- **Cross-Platform Compatibility:** Ensures the game runs consistently across different operating systems

Automated Unit Testing Tool

Python's unittest framework was chosen as the automated testing tool because:

- **Built-in Framework:** No external dependencies required, ensuring project portability
- **Comprehensive Features:** Supports test fixtures, test suites, assertions, and mocking capabilities
- **TDD-Friendly:** Designed to support test-first development with clear pass/fail indicators
- **Integration:** Seamlessly integrates with Python development workflow and IDEs
- **Reporting:** Provides detailed test results and coverage information

---

Process

Test-Driven Development Implementation

The development process strictly followed the TDD red-green-refactor cycle:

1. Red Phase: Writing Failing Tests

Each feature began with writing comprehensive unit tests that initially failed:

Example - Game Initialisation Tests:

```
def test_game_initialization(self):
    """Test game initializes with correct default values"""
    self.assertEqual(self.game.lives, 6)
    self.assertEqual(self.game.guessed_letters, set())
    self.assertFalse(self.game.game_over)
    self.assertFalse(self.game.won)
```

Key Requirements Tested:

- Game state initialisation with 6 lives
- Empty guessed letters set
- Correct game over and win conditions
- Word/phrase display formatting
- Timer functionality and timeout handling
- Input validation and error handling

2. Green Phase: Implementing Minimal Code

After writing failing tests, minimal code was implemented to make the tests pass:

Example - Core Game Logic:

```
class HangmanGame:
    def __init__(self):
        self.lives = 6
        self.guessed_letters = set()
        self.game_over = False
        self.won = False
```

3. Refactor Phase: Code Improvement

Once tests passed, code was refactored for better design, performance, and maintainability while ensuring all tests continued to pass.

Automated Unit Testing Tool Usage

Test Structure and Organisation

The test suite was organised into four main test classes:

1. **TestHangmanGame:** Core game logic testing
    - Game initialization
    - Word setting and display
    - Guess processing (correct, incorrect, duplicate)
    - Win/lose condition detection
    - Timeout handling
2. **TestDictionaryManager:** Word management testing
    - Random word selection for different levels
    - Invalid level handling
    - Custom word addition functionality
3. **TestGameTimer:** Timer functionality testing
    - Timer initialisation and state management
    - Start/stop functionality
    - Timeout callback execution
    - Thread safety verification
4. **TestGameInterface:** User interface testing
    - Input validation and sanitisation
    - Level selection processing
    - Display method functionality

Test Coverage and Results

Final Test Statistics:

- Total Tests: 19 comprehensive unit tests
- Pass Rate: 100% (19/19 tests passing)
- Execution Time: ~1.1 seconds
- Coverage Areas: All major functionality covered

Sample Test Execution Output:

```
test_correct_guess ... ok
test_duplicate_guess ... ok
test_game_initialization ... ok
test_incorrect_guess ... ok
test_lose_condition ... ok
test_set_phrase_intermediate_level ... ok
test_set_word_basic_level ... ok
test_timeout_deducts_life ... ok
test_win_condition ... ok
...
Ran 19 tests in 1.113s
OK
```

Code Quality Assurance

Automated Linting Tools

Two automated code quality tools were used throughout development:

1. Flake8 - Style and Error Detection:

- Enforced PEP 8 style guidelines
- Detected syntax errors and undefined variables
- Identified code complexity issues
- Final Result: 0 warnings/errors

2. Pylint - Comprehensive Code Analysis:

- Analysed code structure and design
- Detected potential bugs and code smells
- Enforced naming conventions
- Measured code complexity and maintainability
- Final Score: 10.00/10 (Perfect Score)

Code Quality Improvements Made

Issues Addressed:

- Removed 121+ trailing whitespace instances
- Added missing final newlines to all files
- Fixed line length violations (>79 characters)
- Added comprehensive docstrings for all modules and classes
- Removed unused imports and variables
- Improved exception handling specificity
- Eliminated unnecessary code constructs

Requirements Implementation

Software Requirements

All Necessary Requirements Met:

- Two difficulty levels (Basic: words, Intermediate: phrases)
- Timer system with 15-second countdown
- Life management system (6 lives)
- Input validation and error handling
- Win/lose condition detection
- Game state persistence and reset functionality

User Interface:

- Clean console-based interface
- Clear game state display with emoji indicators
- Intuitive menu system
- Real-time timer display
- Comprehensive error messages and user guidance

Coding Standards

Code Design:

- Modular architecture with single responsibility principle
- Object-oriented design with clear class hierarchies
- Separation of concerns between game logic, UI, and data management
- Thread-safe timer implementation

Coding Style:

- PEP 8 compliant formatting
- Meaningful variable and function names
- Consistent indentation and spacing

- Appropriate use of Python language features

Documentation:

- Comprehensive module docstrings
- Detailed method documentation with parameters and return types
- Clear inline comments explaining complex logic
- Professional README with usage instructions

Compilation and Execution:

- Error-free compilation and execution
- Robust error handling prevents crashes
- Graceful degradation for edge cases
- Cross-platform compatibility

Code Quality Tools:

- All Flake8 warnings addressed (0 remaining)
- Perfect Pylint score achieved (10.00/10)
- Continuous integration of quality checks throughout development

---

Conclusion

Lessons Learned

What Went Well

1. Test-Driven Development Effectiveness:

- TDD methodology provided clear development direction and confidence in code correctness
- Writing tests first helped identify edge cases early in development
- The red-green-refactor cycle ensured continuous code improvement
- Comprehensive test coverage (19 tests) caught numerous bugs during development

2. Code Quality Achievement:

- Achieved perfect code quality scores (Pylint 10.00/10, Flake8 0 warnings)
- Automated linting tools provided consistent feedback and maintained high standards
- Modular design made the codebase maintainable and extensible
- Clear documentation enhanced code readability and future maintenance

3. Feature Implementation Success:

- All required features implemented successfully (timer, lives, difficulty levels)
- User interface provides clear feedback and intuitive interaction
- Error handling ensures robust operation under various conditions
- Game logic handles all edge cases correctly

Areas for Improvement

1. User Experience Enhancements:

- Visual Improvements: Add ASCII art hangman drawing for visual feedback
- Color Support: Implement colored console output for better user experience
- Sound Effects: Add audio feedback for correct/incorrect guesses
- Progress Indicators: Visual progress bars for timer and game completion

2. Feature Expansions:

- Hint System: Provide optional hints for difficult words/phrases
- Score Tracking: Implement high score system with persistent storage
- Multiplayer Mode: Add support for multiple players taking turns
- Custom Dictionaries: Allow users to load custom word lists from files

3. Technical Improvements:

- Configuration Management: External configuration file for game settings
- Logging System: Implement comprehensive logging for debugging and analytics
- Performance Optimization: Optimize timer thread management for better resource usage
- Database Integration: Store game statistics and user preferences

How Improvements Can Be Implemented

Short-term Improvements (1-2 weeks):

1. Visual Enhancements: Implement colorama library for colored console output
2. ASCII Art: Add hangman drawing that updates with each wrong guess
3. Configuration File: Create JSON configuration for customizable game settings
4. Extended Dictionary: Expand word lists with themed categories...

Medium-term Improvements (1-2 months):

1. GUI Version: Develop tkinter-based graphical user interface
2. Database Integration: Implement SQLite for persistent score tracking
3. Web Version: Create Flask-based web application version
4. Mobile Compatibility: Develop responsive web interface for mobile devices

Long-term Improvements (3-6 months):

1. Multiplayer Network Game: Implement client-server architecture for online multiplayer
2. AI Integration: Add AI-powered hint system and difficulty adjustment
3. Cross-Platform Mobile App: Develop native mobile applications
4. Analytics Dashboard: Create comprehensive game analytics and reporting system

Project Success Metrics

Technical Achievements:

- Perfect code quality score (Pylint 10.00/10)
- Zero linting warnings (Flake8 clean)
- 100% test pass rate (19/19 tests)
- Complete feature implementation
- Robust error handling and edge case coverage

Development Process Success:

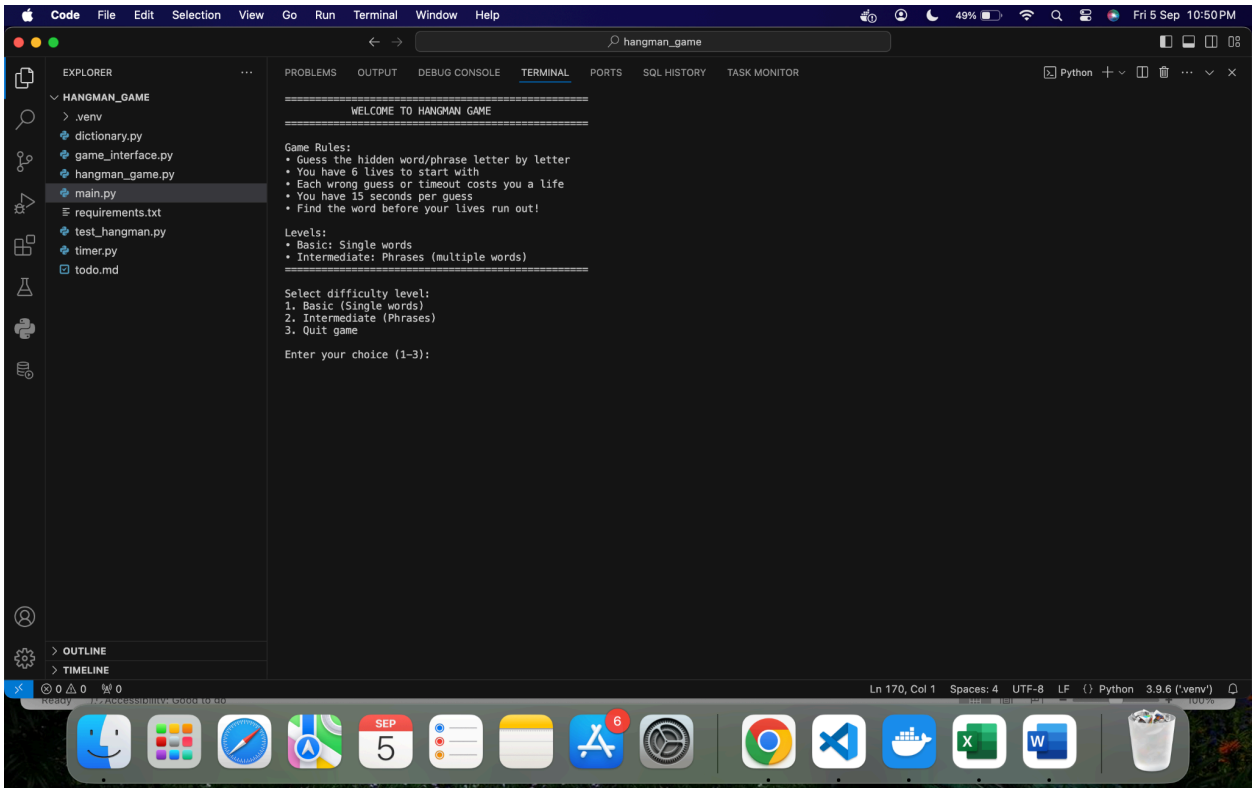
- Successful TDD implementation throughout development
- Comprehensive documentation and code comments
- Modular, maintainable code architecture
- Professional-grade deliverables (README, tests, documentation)

GitHub Repository

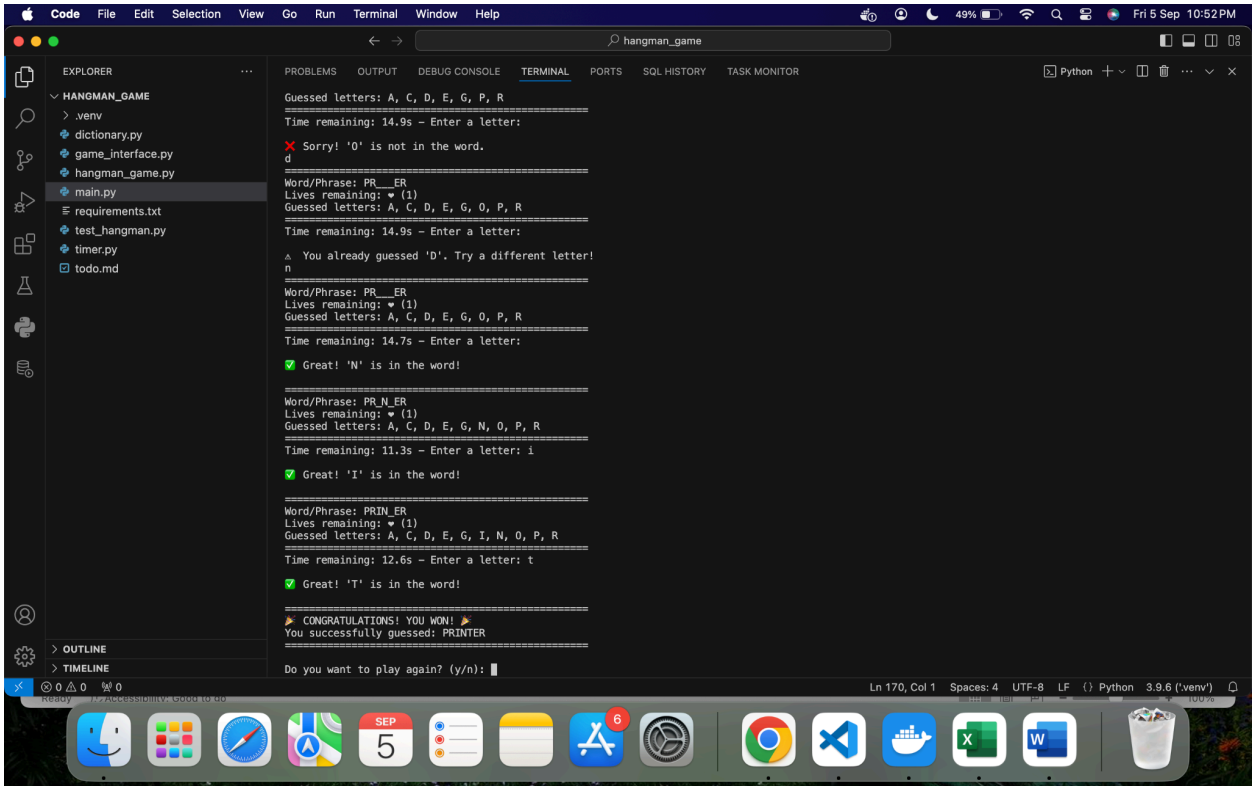
The complete project, including all source code, tests, documentation, and this report, is available in the GitHub repository. The repository demonstrates professional software development practices and serves as a comprehensive example of TDD implementation in Python.

Repository Link:
https://github.com/digitalaashish/hangman

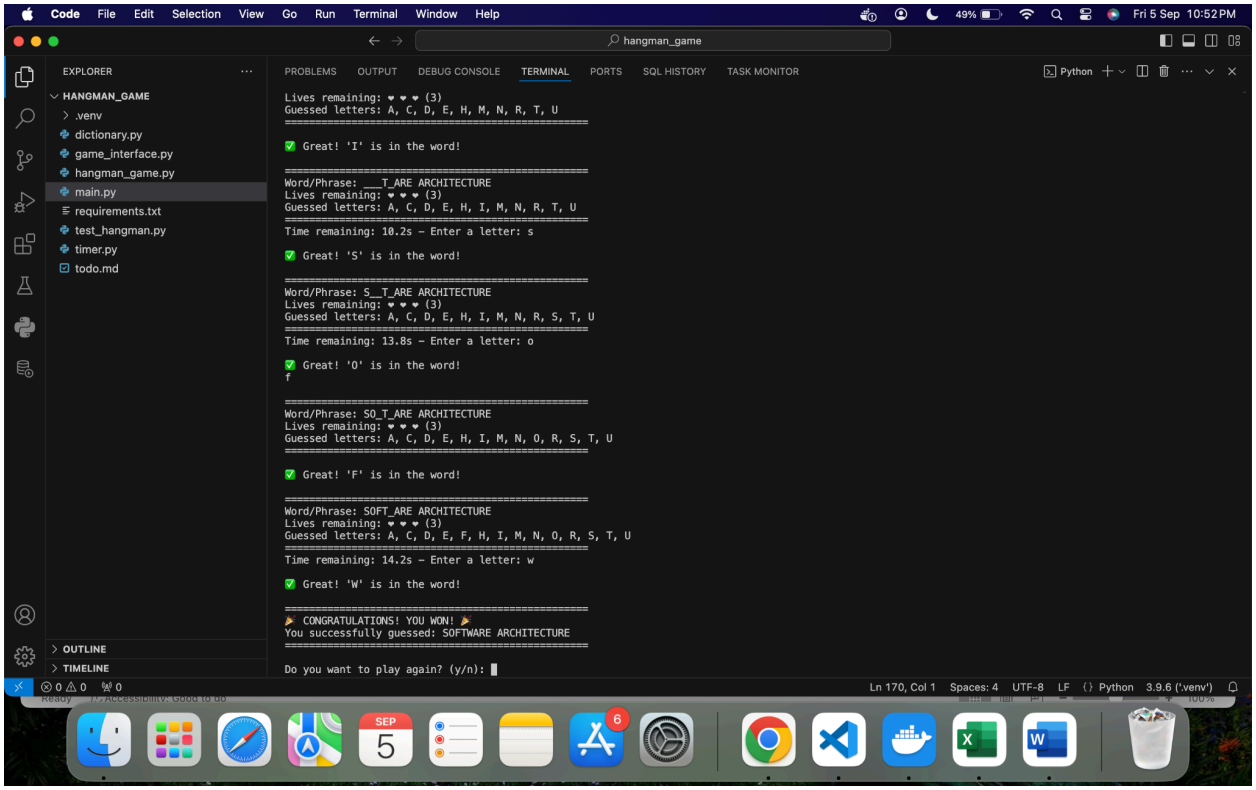
Screenshots



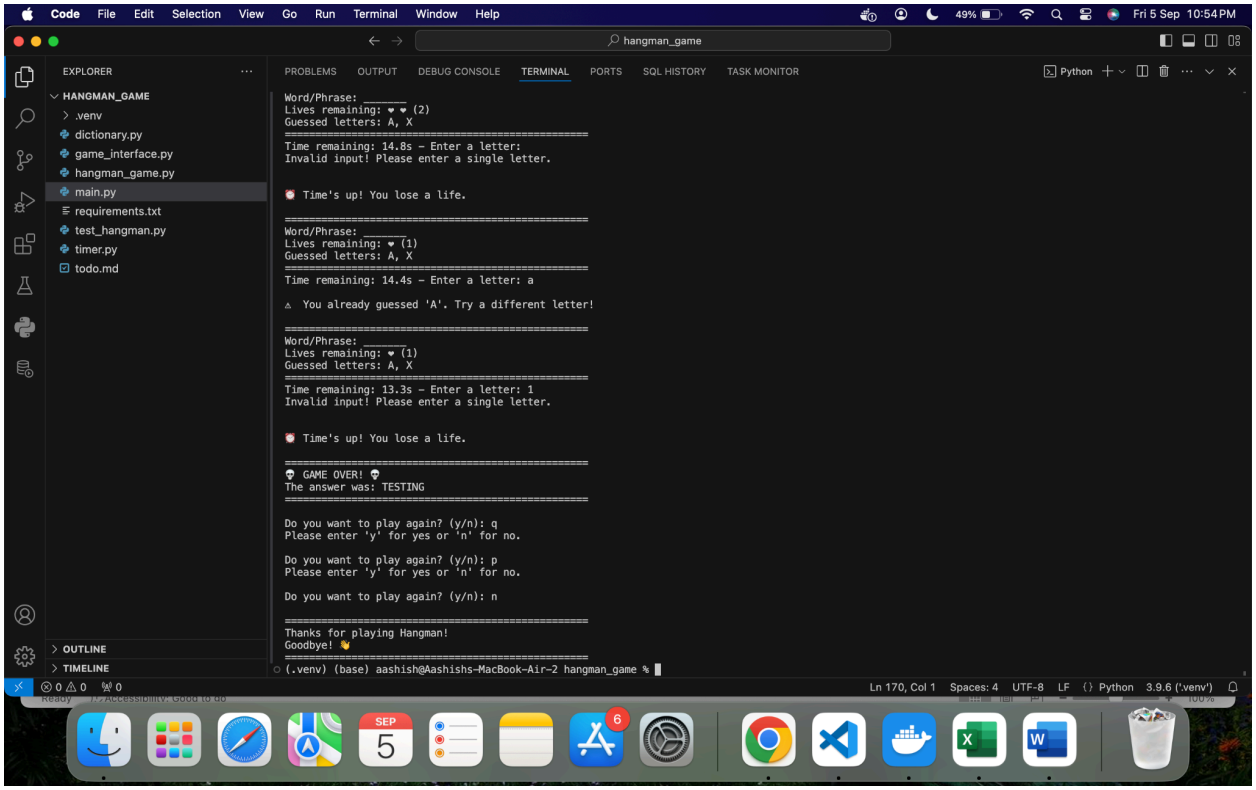
Main file to start the game



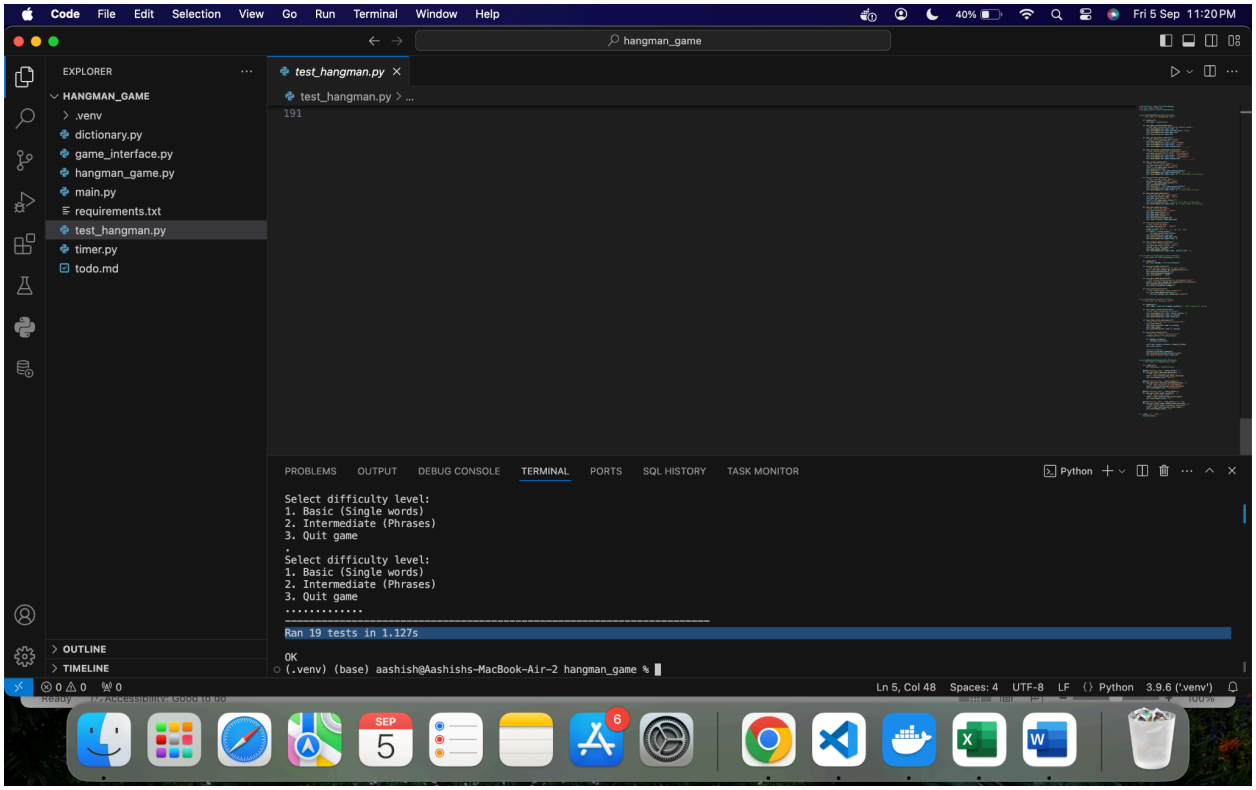
You won the game (basic)



You won the game (Intermediate)



Times Up! You lose a life, Game Over!!!



Testing Hangman….

This report demonstrates the successful implementation of a Hangman game using Test-Driven Development methodology, achieving perfect code quality standards and comprehensive feature implementation.