Charles Darwin University

(SYDNEY CAMPUS)

S225 PRT582

SOFTWARE ENGINEERING: PROCESS AND TOOLS

Software Unit Testing Report

Assignment 2

Name : Md Ahnaf Rashid

Student ID: 394051 (CDU SYDNEY CAMPUS)

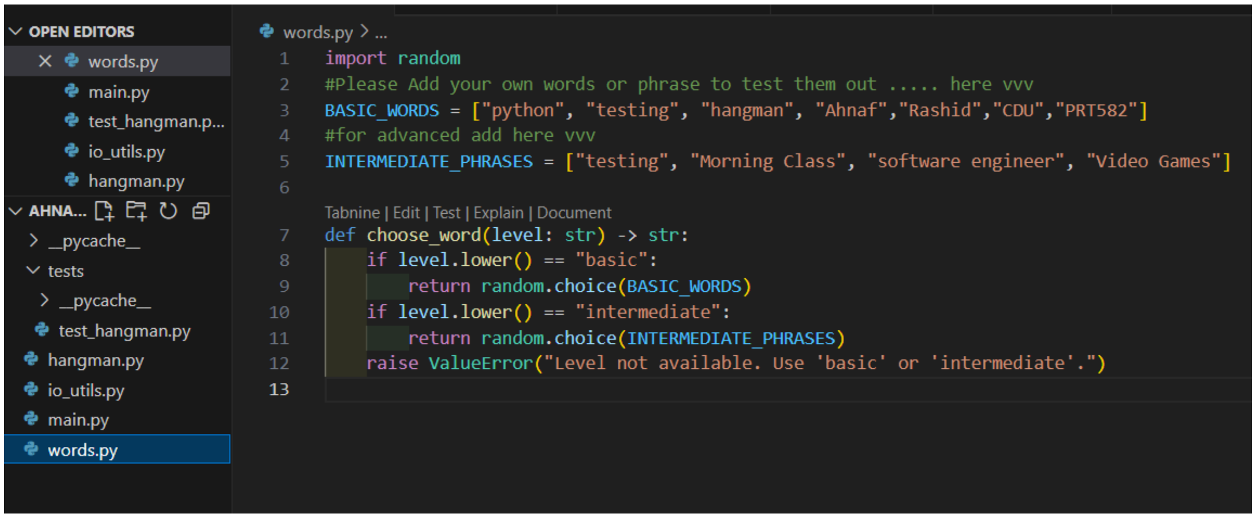
Email: s394051@students.cdu.edu.au

Software Unit Testing Report: Hangman Game

Introduction

The objective here was to develop a Hangman game using Test-Driven Development (TDD) by using Python. Hangman is a famous word-guessing game where players attempt to find a hidden word and, In our case, also a phrase by guessing letters within a limited number of life number which limits our attempts or chances. According to the instructions given to us the game has two difficulty levels: "basic" for single words and "intermediate" for phrases, which can easily be set in the following file here:

In **words.py** file



```python
import random
#Please Add your own words or phrase to test them out ..... here vvv
BASIC_WORDS = ["python", "testing", "hangman", "Ahnaf","Rashid","CDU","PRT582"]
#for advanced add here vvv
INTERMEDIATE_PHRASES = ["testing", "Morning Class", "software engineer", "Video Games"]

def choose_word(level: str) -> str:
    if level.lower() == "basic":
        return random.choice(BASIC_WORDS)
    if level.lower() == "intermediate":
        return random.choice(INTERMEDIATE_PHRASES)
    raise ValueError("Level not available. Use 'basic' or 'intermediate'.")
```

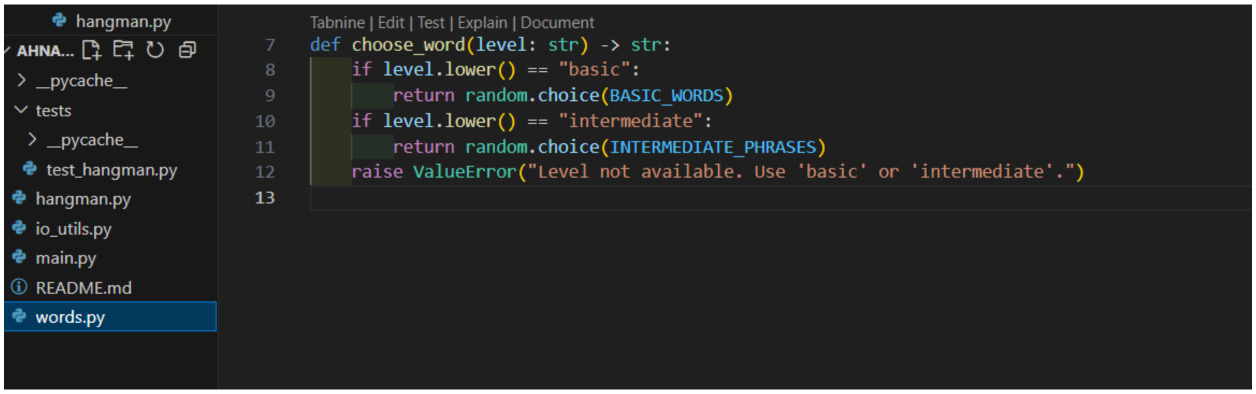
We are also given a time a time limit of 15 seconds where if not guessed by that time we will start losing life points as well. Now python was used because of its ease of use and ready to use libraries available which made the whole development process faster and testing smoother.

Process

The test driven development process required us to first try and test out possible solutions and find their results if they were able to meet our requirements. First of the must haves of our code were noted and kept in mind and then functions and other components were added so that the TDD can proceed smoothly. Below, each requirement is addressed, with explanations of how TDD and testing were applied, supported by screenshots.

Requirement : Two Levels (Basic and Intermediate)

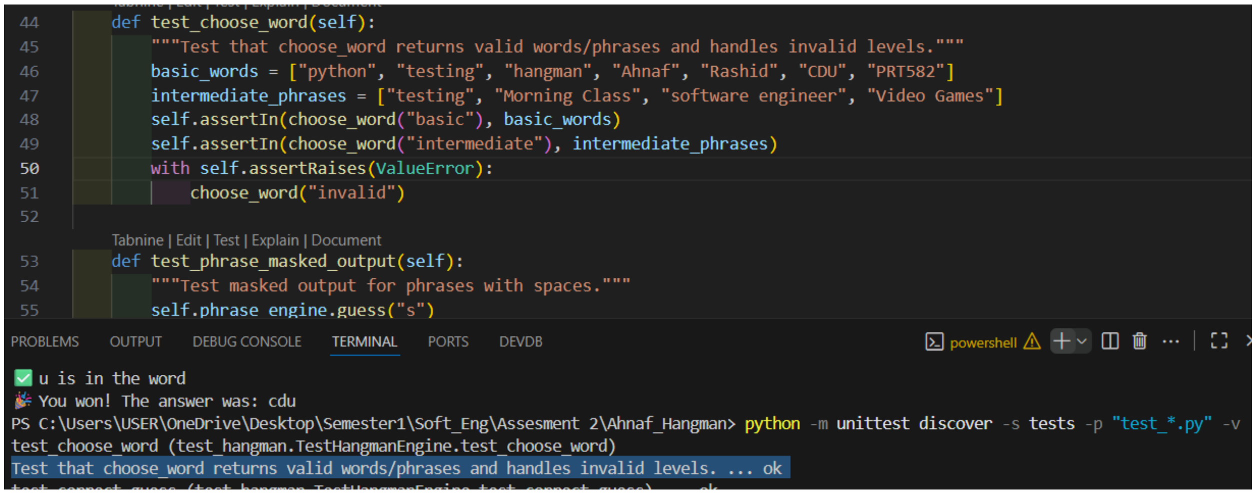
The game again as told supports two levels: "basic" for single words and "intermediate" for phrases, implemented in words.py it checks the equal condition and when it matches the level it takes up the word from dictionary to guess the word else prints the ValueError part with the message “Level Is not available …….”)



```python
def choose_word(level: str) -> str:
    if level.lower() == "basic":
        return random.choice(BASIC_WORDS)
    if level.lower() == "intermediate":
        return random.choice(INTERMEDIATE_PHRASES)
    raise ValueError("Level not available. Use 'basic' or 'intermediate'.")
```

Requirement : Valid Words from a Dictionary

Words and phrases are taken from already available word lists in words.py, acting as a simplified dictionary like python for basic words and software engineer for intermediate where these are actually phrases. The test_choose_word test confirms that selected words or phrases is valid by checking whether or not they belong to the defined and already placed words in the lists. The test_phrase_masked_output test verifies phrase handling here. Note that the test function works like a mirror for words.py file where the actual words are stored.



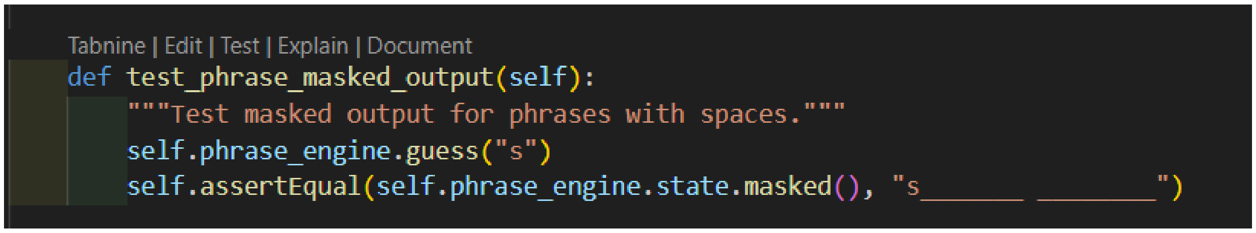
```
def test_choose_word(self):
    """Test that choose_word returns valid words/phrases and handles invalid levels."""
    basic_words = ["python", "testing", "hangman", "Ahnaf", "Rashid", "CDU", "PRT582"]
    intermediate_phrases = ["testing", "Morning Class", "software engineer", "Video Games"]
    self.assertIn(choose_word("basic"), basic_words)
    self.assertIn(choose_word("intermediate"), intermediate_phrases)
    with self.assertRaises(ValueError):
        choose_word("invalid")

Tabnine | Edit | Test | Explain | Document
def test_phrase_masked_output(self):
    """Test masked output for phrases with spaces."""
    self.phrase_engine.guess("s")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS DEVDB powershell

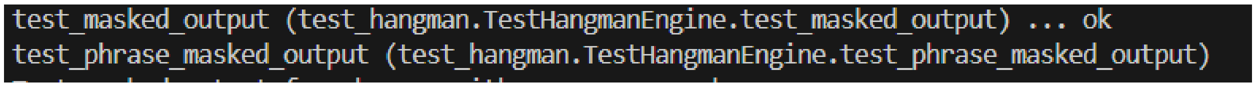
```
✅ u is in the word
🎉 You won! The answer was: cdu
PS C:\Users\USER\OneDrive\Desktop\Semester1\Soft_Eng\Assesment 2\Ahnaf_Hangman> python -m unittest discover -s tests -p "test_*.py" -v
test_choose_word (test_hangman.TestHangmanEngine.test_choose_word)
Test that choose_word returns valid words/phrases and handles invalid levels. ... ok
```

Requirement : Underscores for Missing Letters

The game displays underscores (_) for the missing letters which are to be guessed, implemented in HangmanState.masked. The test_masked_output test here confirms that guessing "t" in "test" produces "t__t". That is once we have guessed one word all the other places where the same letter appeared will be filled up automatically. The test_phrase_masked_output test makes phrases like "software engineer" display correctly (e.g., "s_______ ________" after guessing "s"), and also handles the spaces properly.



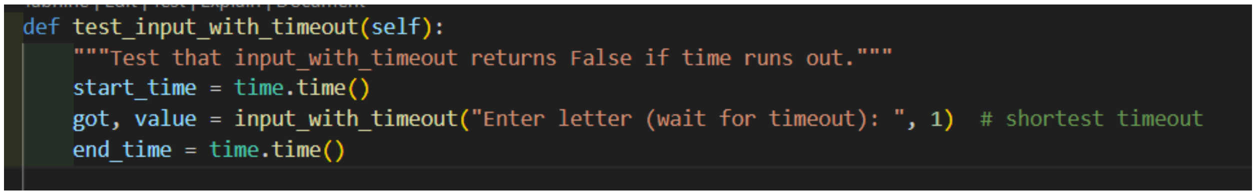
```
Tabnine | Edit | Test | Explain | Document
def test_phrase_masked_output(self):
    """Test masked output for phrases with spaces."""
    self.phrase_engine.guess("s")
    self.assertEqual(self.phrase_engine.state.masked(), "s_______ ________")
```



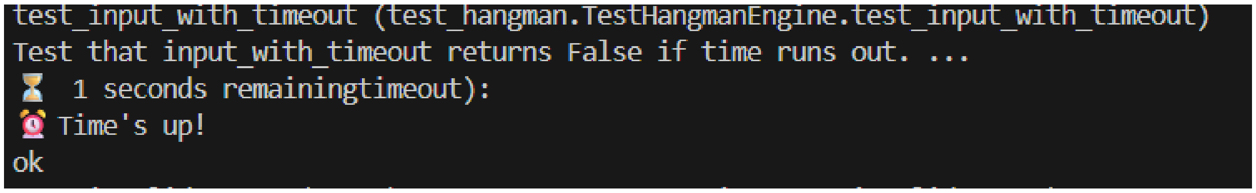
```
test_masked_output (test_hangman.TestHangmanEngine.test_masked_output) ... ok
test_phrase_masked_output (test_hangman.TestHangmanEngine.test_phrase_masked_output)
```

Requirement : 15-Second Timer with Life Deduction

Players have 15 seconds to guess, with a countdown displayed, and a life is deducted if time runs out, implemented in io_utils.py’s input_with_timeout and main.py. The test_input_with_timeout in thetest_hangman file confirms that



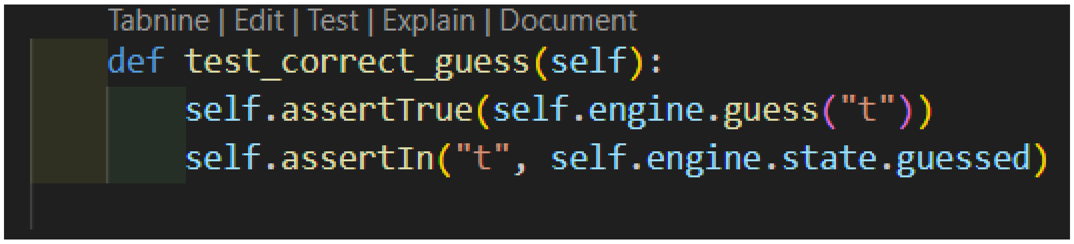
```
def test_input_with_timeout(self):
    """Test that input_with_timeout returns False if time runs out."""
    start_time = time.time()
    got, value = input_with_timeout("Enter letter (wait for timeout): ", 1)  # shortest timeout
    end_time = time.time()
```



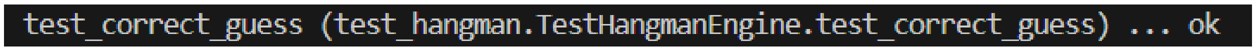
```
test_input_with_timeout (test_hangman.TestHangmanEngine.test_input_with_timeout)
Test that input_with_timeout returns False if time runs out. ...
⏳  1 seconds remainingtimeout):
⏰ Time's up!
ok
```

Requirement : Reveal All Instances of a Guessed Letter

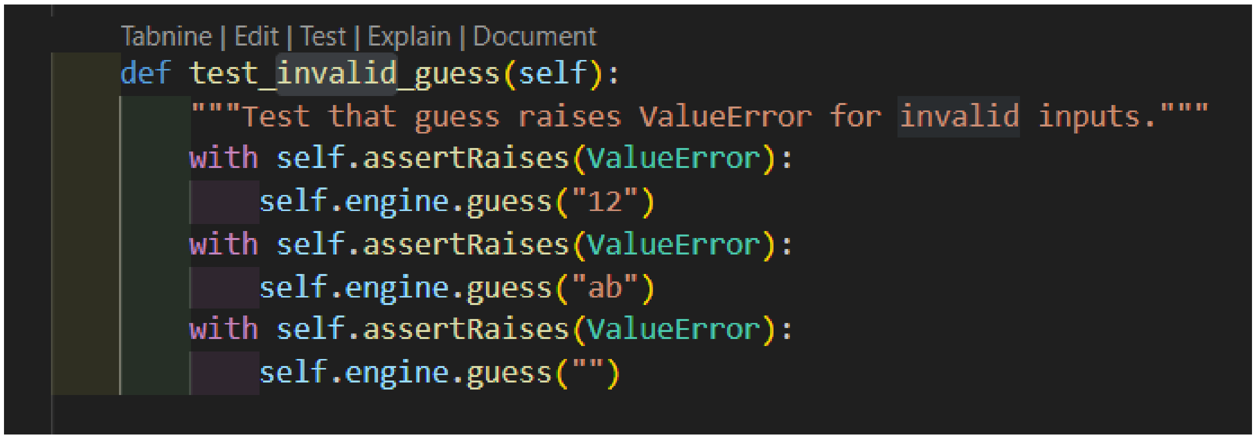
Correctly guesses and reveal all occurrences of that alphabet, implemented in HangmanEngine.guess. The test_correct_guess test verifies that guessing "t" returns True and adds "t" to the guessed set. The test_masked_output test confirms all instances of "t" appear in "t__t" for "test". The test_invalid_guess test ensures invalid inputs (example, "12") raise a ValueError.



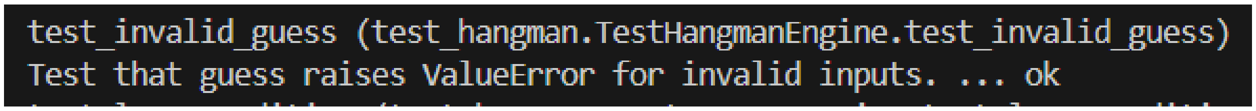
```
def test_correct_guess(self):
    self.assertTrue(self.engine.guess("t"))
    self.assertIn("t", self.engine.state.guessed)
```



```
test_correct_guess (test_hangman.TestHangmanEngine.test_correct_guess) ... ok
```



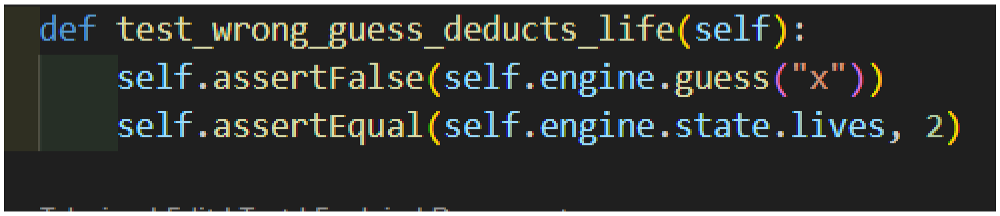
```
def test_invalid_guess(self):
    """Test that guess raises ValueError for invalid inputs."""
    with self.assertRaises(ValueError):
        self.engine.guess("12")
    with self.assertRaises(ValueError):
        self.engine.guess("ab")
    with self.assertRaises(ValueError):
        self.engine.guess("")
```



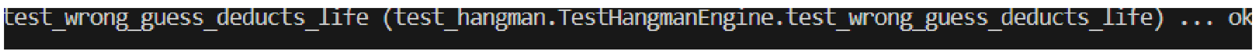
```
test_invalid_guess (test_hangman.TestHangmanEngine.test_invalid_guess)
Test that guess raises ValueError for invalid inputs. ... ok
```

Requirement : Life Deduction for Wrong Guesses

Incorrect guesses deduct a life, implemented in HangmanEngine.guess. The test_wrong_guess_deducts_life test confirms that guessing "x" (not in "test") returns False and reduces lives from 3 to 2.



```python
def test_wrong_guess_deducts_life(self):
    self.assertFalse(self.engine.guess("x"))
    self.assertEqual(self.engine.state.lives, 2)
```



```
test_wrong_guess_deducts_life (test_hangman.TestHangmanEngine.test_wrong_guess_deducts_life) ... ok
```

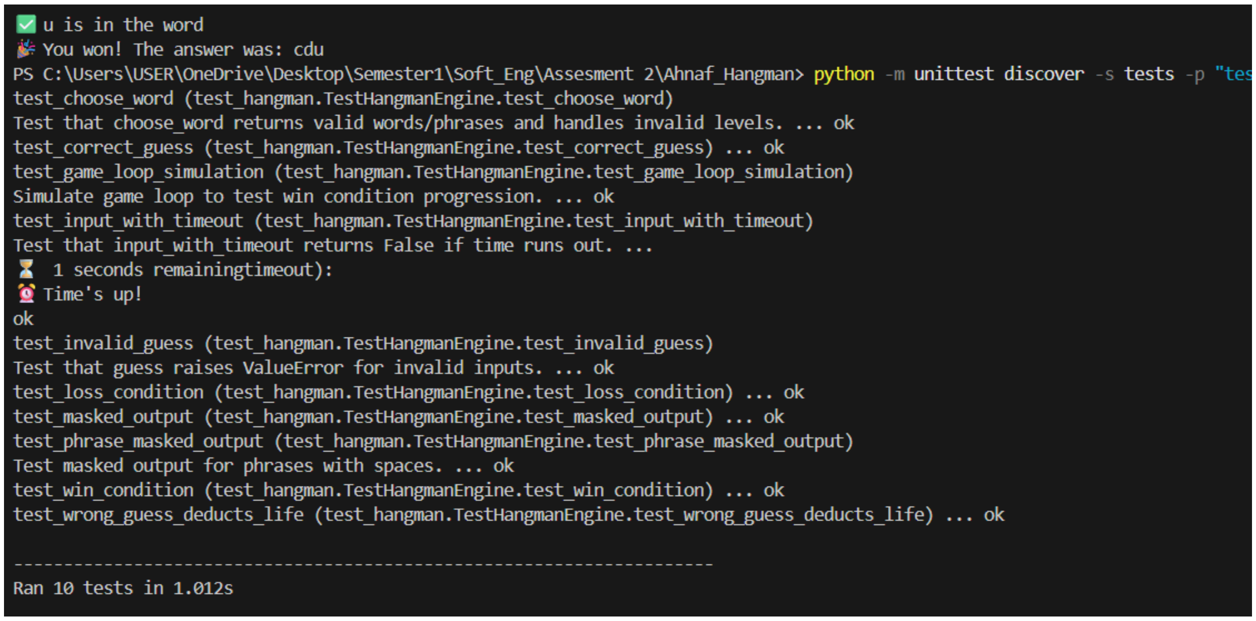
Requirement : Find Word Before Lives Reach Zero

The player must guess the word before lives reach zero. The test_win_condition test verifies that guessing "t", "e", and "s" in "test" results in a win (is_won() returns True). The test_loss_condition test confirms that guessing "x", "y", and "z" until lives reach zero results in a loss (is_lost() returns True).

Requirement : Game Continuation and Answer Display

The game continues until the player wins, loses, or quits (via timeout/loss), and the answer is displayed at the end, implemented in main.py's run_game. The test_game_loop_simulation test simulates guessing letters ("c", "a", "t" for "cat") to reach a win state, verifying game progression. It has been manually tested.

All tests have passed as shown below



```
✅ u is in the word
🎉 You won! The answer was: cdu
PS C:\Users\USER\OneDrive\Desktop\Semester1\Soft_Eng\Assesment 2\Ahnaf_Hangman> python -m unittest discover -s tests -p "tes
test_choose_word (test_hangman.TestHangmanEngine.test_choose_word)
Test that choose_word returns valid words/phrases and handles invalid levels. ... ok
test_correct_guess (test_hangman.TestHangmanEngine.test_correct_guess) ... ok
test_game_loop_simulation (test_hangman.TestHangmanEngine.test_game_loop_simulation)
Simulate game loop to test win condition progression. ... ok
test_input_with_timeout (test_hangman.TestHangmanEngine.test_input_with_timeout)
Test that input_with_timeout returns False if time runs out. ...
⌛ 1 seconds remainingtimeout):
⏰ Time's up!
ok
test_invalid_guess (test_hangman.TestHangmanEngine.test_invalid_guess)
Test that guess raises ValueError for invalid inputs. ... ok
test_loss_condition (test_hangman.TestHangmanEngine.test_loss_condition) ... ok
test_masked_output (test_hangman.TestHangmanEngine.test_masked_output) ... ok
test_phrase_masked_output (test_hangman.TestHangmanEngine.test_phrase_masked_output)
Test masked output for phrases with spaces. ... ok
test_win_condition (test_hangman.TestHangmanEngine.test_win_condition) ... ok
test_wrong_guess_deducts_life (test_hangman.TestHangmanEngine.test_wrong_guess_deducts_life) ... ok

----------------------------------------------------------------------
Ran 10 tests in 1.012s
```

TDD Process

Test driven development was applied by writing tests in test_hangman.py before implementing features in hangman.py, io_utils.py, words.py, and main.py. For each requirement:

1. A test was written to find out what we expected and deduce how it will behave
2. The test failed initially due to not being able to come up with scenarios where errors might occur at first
3. Code was written and adjusted to pass a few tests.

Conclusion

In conclusion the TDD method and the unittest method were used to create the Hangman game, and tests in test_hangman.py gave the output where all conditions were satisfied. By creating failing tests first and then developing based on that gradually with minimum code to pass the bare minimum requirements that are necessary made the TDD process a dependable development process. Managing gaps in phrases (fixed in test_phrase_masked_output) and using threading to implement the timer were some minor challenges faced during the development and some tests were done manually as automation was far too complicated. Among the lessons learnt include the necessity of physically verifying console output, the significance of exact test expectations. In the future one can also add GUI or a more applealing graphics to the development to further enhance the game.

GitHub link can be found at Ahnaf-r71/Ahnaf_Hangman_TDDev: Hangman Games Using Test Driven Development