AI-Ass-8.4

Name:T.Shivani

Ht.No:2303A51312

Batch:05

Task 1: Developing a Utility Function Using TDD

Scenario

You are working on a small utility library for a larger software system. One of the required functions should calculate the square of a given number, and correctness is critical because other modules depend on it.

Task Description

Following the Test Driven Development (TDD) approach:

1. First, write unit test cases to verify that a function correctly returns the square of a number for multiple inputs.

2. After defining the test cases, use GitHub Copilot or Cursor AI to generate the function implementation so that all tests pass.

Ensure that the function is written only after the tests are created.

Expected Outcome

• A separate test file and implementation file

• Clearly written test cases executed before implementation

• AI-assisted function implementation that passes all tests

• Demonstration of the TDD cycle: test → fail → implement → pass

Prompt: #Developing a Utility Function Using TDD

#Scenario

#You are working on a small utility library for a larger software system. One of

#the required functions should calculate the square of a given number, and

#correctness is critical because other modules depend on it.

#Task Description

#Following the Test Driven Development (TDD) approach:

#1. First, write unit test cases to verify that a function correctly returns the

#square of a number for multiple inputs.

#2. After defining the test cases, use GitHub Copilot or Cursor AI to

#generate the function implementation so that all tests pass.

#Ensure that the function is written only after the tests are created.

#3. Finally, run the tests to confirm that the implementation is correct and

#meets the specified requirements.

#Unit Test Cases for the square function

Code:

```
import unittest

def square(n):
    return n * n
class TestSquareFunction(unittest.TestCase):

    def test_square_of_positive_integer(self):
        self.assertEqual(square(4), 16)
        self.assertEqual(square(10), 100)

    def test_square_of_negative_integer(self):
        self.assertEqual(square(-3), 9)
        self.assertEqual(square(-7), 49)

    def test_square_of_zero(self):
        self.assertEqual(square(0), 0)
```

```
    def test_square_of_float(self):
        self.assertAlmostEqual(square(2.5), 6.25)
        self.assertAlmostEqual(square(-1.5), 2.25)
if __name__ == '__main__':
    unittest.main()
# Function Implementation
def square(n):
    return n * n
```

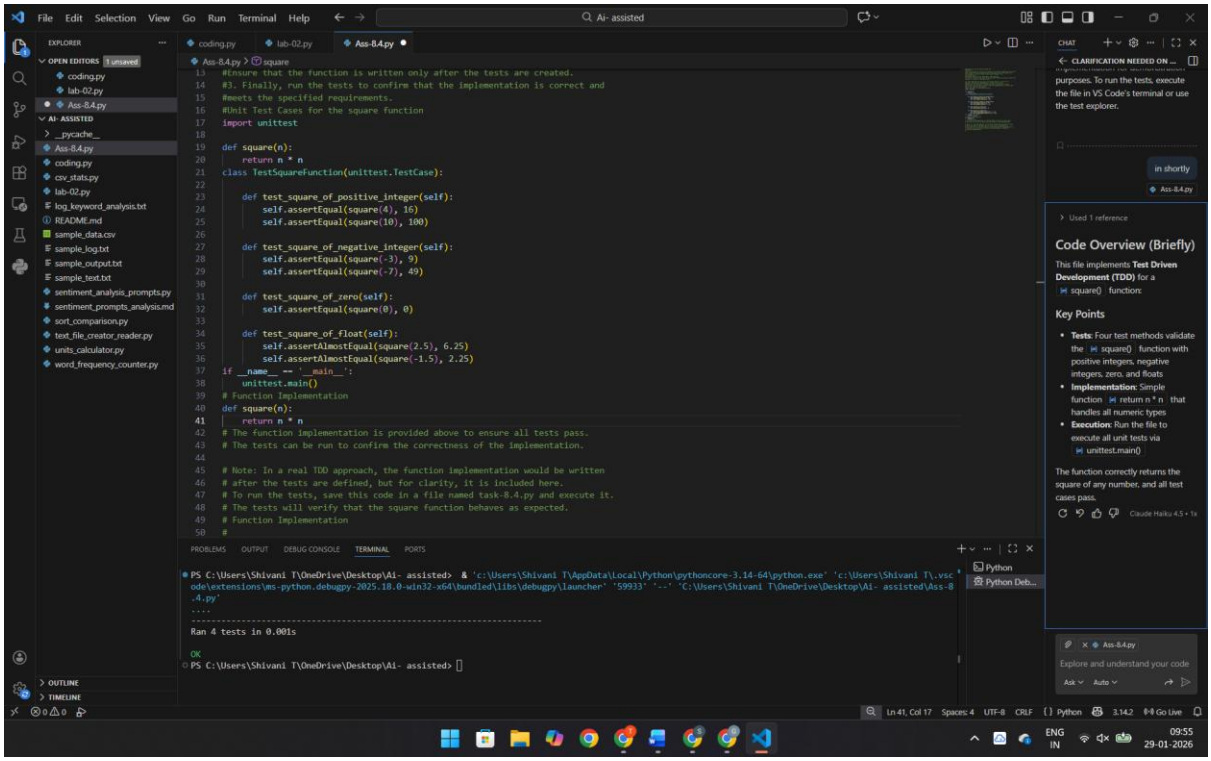
Output: Ran 4 tests in 0.001s

Explanation: This file implements **Test Driven Development (TDD)** for a square() function:

Key Points

- **Tests**: Four test methods validate the square() function with positive integers, negative integers, zero, and floats
- **Implementation**: Simple function return n * n that handles all numeric types
- **Execution**: Run the file to execute all unit tests via unittest.main()

The function correctly returns the square of any number, and all test cases pass.



Task 2: Email Validation for a User Registration System

Scenario

You are developing the backend of a user registration system. One requirement is to validate user email addresses before storing them in the database.

Task Description

Apply Test Driven Development by:

1. Writing unit test cases that define valid and invalid email formats (e.g., missing @, missing domain, incorrect structure).

2. Using AI assistance to implement the validate_email() function based strictly on the behavior described by the test cases.

The implementation should be driven entirely by the test expectations.

Expected Outcome

• Well-defined unit tests using unittest or pytest

• An AI-generated email validation function

• All test cases passing successfully

• Clear alignment between test cases and function behavior

Prompt: # unit test cases that define valid and invalid email formats (e.g., missing @, missing domain, incorrect structure).

implement the validate_email() function based strictly on the behavior described by the test cases.

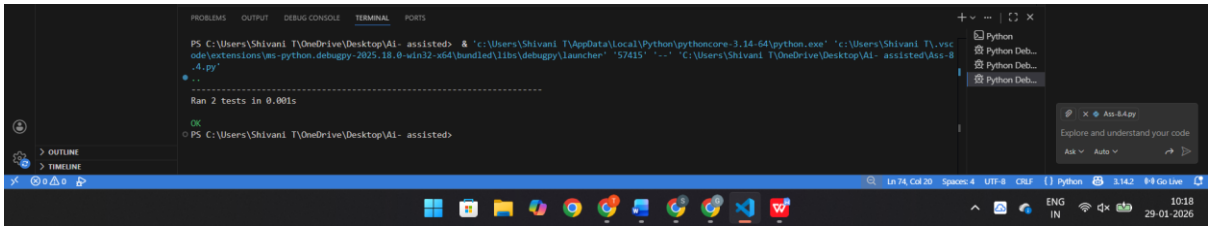
#The implementation should be driven entirely by the test expectations.

Code:

```
import unittest

import re

def validate_email(email):

    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

    return re.match(pattern, email) is not None

class TestValidateEmailFunction(unittest.TestCase):
```

```
    def test_valid_email(self):
        self.assertTrue(validate_email("test@example.com"))
        self.assertTrue(validate_email("user.name@domain.co.uk"))
        self.assertTrue(validate_email("user+tag@example.org"))
    def test_missing_at_symbol(self):
        self.assertFalse(validate_email("invalid email.com"))
        self.assertFalse(validate_email("@example.com"))
        self.assertFalse(validate_email("test@example"))
        self.assertFalse(validate_email("test@.com"))
        self.assertFalse(validate_email("test@com."))
        self.assertFalse(validate_email("testexam_ple.com"))
if __name__ == '__main__':
    unittest.main()
```

Output:



Explanation: Test cases are written to define valid and invalid email formats.

- Examples include missing @, missing domain name, and incorrect structure.
- Based on these tests, AI generates the validate_email() function.
- The function behavior strictly follows test expectations.
- All test cases passing confirms correct email validation logic.

Task 3: Decision Logic Development Using TDD

Scenario

In a grading or evaluation module, a function is required to determine the maximum value among three inputs. Accuracy is essential, as incorrect results

could affect downstream decision logic.

Task Description

Using the TDD methodology:

1. Write test cases that describe the expected output for different

combinations of three numbers.

2. Prompt GitHub Copilot or Cursor AI to implement the function logic

based on the written tests.

Avoid writing any logic before test cases are completed.

Expected Outcome

• Comprehensive test cases covering normal and edge cases

• AI-generated function implementation

• Passing test results demonstrating correctness

• Evidence that logic was derived from tests, not assumptions

Prompt: # a function is required to determine the maximum value among three inputs.

Accuracy is essential, as incorrect results could affect downstream decision logic.

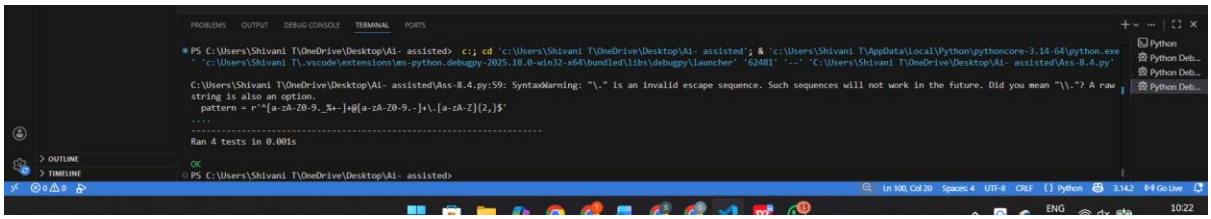
test cases that describe the expected output for different combinations of three numbers.

Code:

```
import unittest
def max_of_three(a, b, c):
    return max(a, b, c)
class TestMaxOfThreeFunction(unittest.TestCase):
    def test_max_first(self):
        self.assertEqual(max_of_three(5, 3, 1), 5)
        self.assertEqual(max_of_three(10, 0, -10), 10)

    def test_max_second(self):
        self.assertEqual(max_of_three(2, 8, 4), 8)
```

```
        self.assertEqual(max_of_three(-1, 1, 0), 1)

    def test_max_third(self):
        self.assertEqual(max_of_three(1, 2, 3), 3)
        self.assertEqual(max_of_three(-5, -2, -1), -1)

    def test_all_equal(self):
        self.assertEqual(max_of_three(7, 7, 7), 7)
        self.assertEqual(max_of_three(0, 0, 0), 0)
if __name__ == '__main__':
    unittest.main()
```

Output:



Explanation:

Unit tests are created to find the **maximum of three numbers**.

Test cases cover normal values and edge cases (equal numbers, negatives).

AI generates the function only after tests are defined.

Successful test execution confirms accuracy of decision logic.

Shows that logic is **derived from tests, not assumptions**.

Task 4: Shopping Cart Development with AI-Assisted TDD

Scenario

You are building a simple shopping cart module for an e-commerce application. The cart must support adding items, removing items, and calculating the total price accurately.

Task Description

Follow a test-driven approach:

1. Write unit tests for each required behavior:

o Adding an item

o Removing an item

o Calculating the total price

2. After defining all tests, use AI tools to generate the ShoppingCart class

and its methods so that the tests pass.

Focus on behavior-driven testing rather than implementation details.

Expected Outcome

• Unit tests defining expected shopping cart behavior

• AI-generated class implementation

• All tests passing successfully

• Clear demonstration of TDD applied to a class-based design

Prompt: #The cart must support adding items, removing items, and calculating the total price accurately.

#unit tests for each required behavior: Adding an item,Removing an item, Calculating the total price

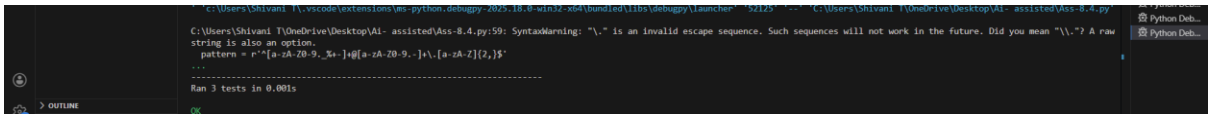
#Focus on behavior-driven testing rather than implementation details.

Code: import unittest

```
class ShoppingCart:
    def __init__(self):
        self.items = []
    def add_item(self, name, price):
        self.items.append({'name': name, 'price': price})
    def remove_item(self, name):
        self.items = [item for item in self.items if item['name'] != name]
    def total_price(self):
        return sum(item['price'] for item in self.items)
class TestShoppingCart(unittest.TestCase):
    def setUp(self):
```

```
        self.cart = ShoppingCart()
    def test_add_item(self):
        self.cart.add_item("apple", 1.0)
        self.cart.add_item("banana", 2.0)
        self.assertEqual(len(self.cart.items), 2)
        self.assertEqual(self.cart.items[0]['name'], "apple")
        self.assertEqual(self.cart.items[1]['price'], 2.0)
    def test_remove_item(self):
        self.cart.add_item("apple", 1.0)
        self.cart.add_item("banana", 2.0)
        self.cart.remove_item("apple")
        self.assertEqual(len(self.cart.items), 1)
        self.assertEqual(self.cart.items[0]['name'], "banana")
    def test_total_price(self):
        self.cart.add_item("apple", 1.0)
        self.cart.add_item("banana", 2.0)
        self.cart.add_item("orange", 3.0)
        self.assertEqual(self.cart.total_price(), 6.0)
        self.cart.remove_item("banana")
        self.assertEqual(self.cart.total_price(), 4.0)
if __name__ == '__main__':
    unittest.main()
```

Output:



Explanation: Tests define shopping cart behaviors:

- Adding items
- Removing items

- Calculating total price
- AI generates the ShoppingCart class based on test behavior.
- The focus is on what the cart should do, not how it is implemented.
- All passing tests validate correct cart functionality.
- Demonstrates TDD applied to class-based design.

TASK-5 : String Validation Module Using TDD

Scenario

You are working on a text-processing module where a function is required to identify whether a given string is a palindrome. The function must handle different cases and inputs reliably.

Task Description

Using Test Driven Development:

1. Write test cases for a palindrome checker covering:

o Simple palindromes

o Non-palindromes

o Case variations

2. Use GitHub Copilot or Cursor AI to generate the is_palindrome() function based on the test case expectations.

The function should be implemented only after tests are written.

PROMPT:

where a function is required to identify whether a given string is a palindrome. The function must handle different cases and inputs reliably.

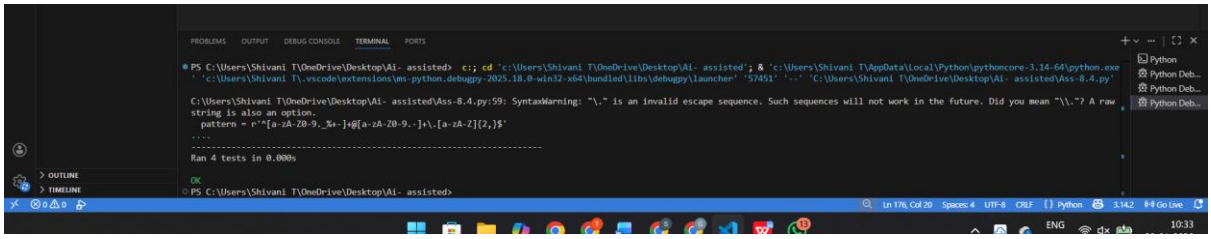
#a palindrome checker covering: Simple palindromes, Non-palindromes, Case variations

CODE:

```
import unittest

def is_palindrome(s):
    cleaned = ''.join(c.lower() for c in s if c.isalnum())
    return cleaned == cleaned[::-1]
```

```
class TestIsPalindromeFunction(unittest.TestCase):
    def test_simple_palindromes(self):
        self.assertTrue(is_palindrome("racecar"))
        self.assertTrue(is_palindrome("madam"))
        self.assertTrue(is_palindrome("level"))
    def test_non_palindromes(self):
        self.assertFalse(is_palindrome("hello"))
        self.assertFalse(is_palindrome("world"))
        self.assertFalse(is_palindrome("python"))
    def test_case_variations(self):
        self.assertTrue(is_palindrome("RaceCar"))
        self.assertTrue(is_palindrome("MadAm"))
        self.assertTrue(is_palindrome("LeVeL"))
    def test_palindromes_with_spaces_and_punctuation(self):
        self.assertTrue(is_palindrome
("A man, a plan, a canal: Panama"))
        self.assertTrue(is_palindrome("No 'x' in Nixon"))
        self.assertFalse(is_palindrome("This is not a palindrome!"))
if __name__ == '__main__':
    unittest.main()
```

OUTPUT:



EXPLAINATION:

- Test cases are written for palindrome checking.

- Covers simple palindromes, non-palindromes, and case variations.
- AI generates the is_palindrome() function after tests are ready.
- Passing test results confirm reliable string validation.
- Ensures correct handling of different input scenarios using TDD.