ASSIGNMENT 8.4

D.CHARMI

2303A51314

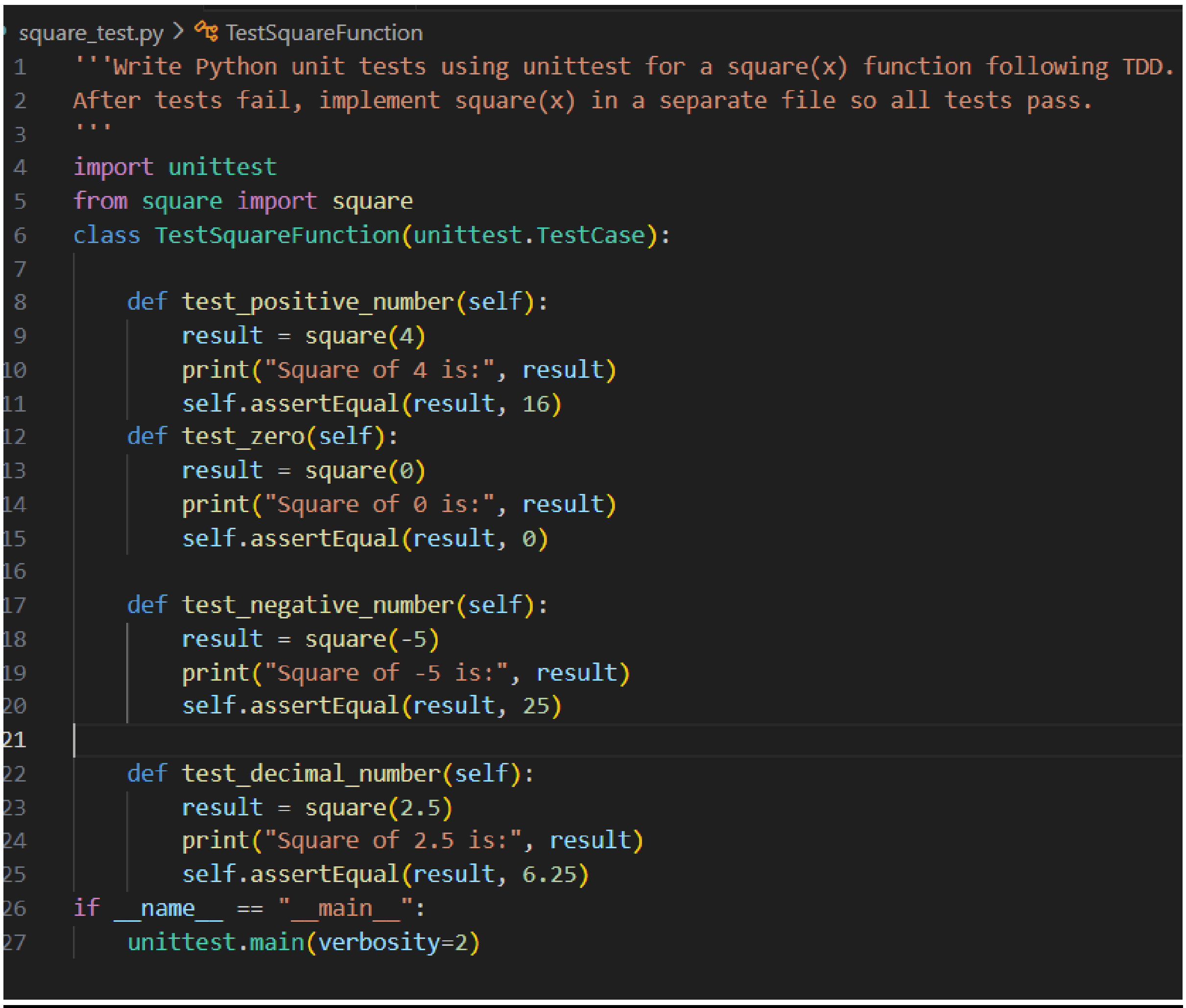
BATCH – 05

TASK – 01

Prompt : Write Python unit tests using unit test for a square(x) function following TDD. After tests fail, implement square(x) in a separate file so all tests pass.

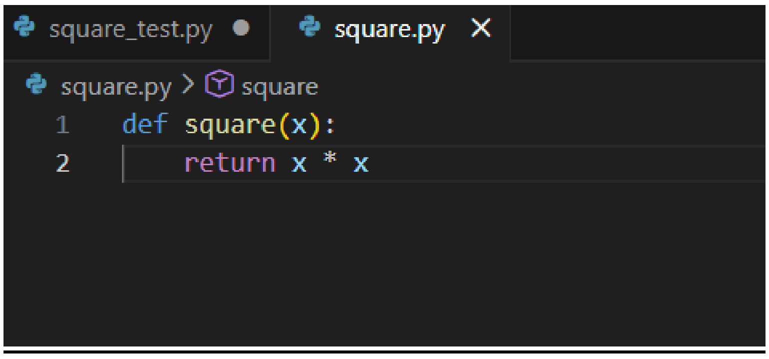
Code:

<u>Square_test.py</u>



```python
'''Write Python unit tests using unittest for a square(x) function following TDD.
After tests fail, implement square(x) in a separate file so all tests pass.
'''
import unittest
from square import square
class TestSquareFunction(unittest.TestCase):

    def test_positive_number(self):
        result = square(4)
        print("Square of 4 is:", result)
        self.assertEqual(result, 16)
    def test_zero(self):
        result = square(0)
        print("Square of 0 is:", result)
        self.assertEqual(result, 0)

    def test_negative_number(self):
        result = square(-5)
        print("Square of -5 is:", result)
        self.assertEqual(result, 25)

    def test_decimal_number(self):
        result = square(2.5)
        print("Square of 2.5 is:", result)
        self.assertEqual(result, 6.25)
if __name__ == "__main__":
    unittest.main(verbosity=2)
```

square.py

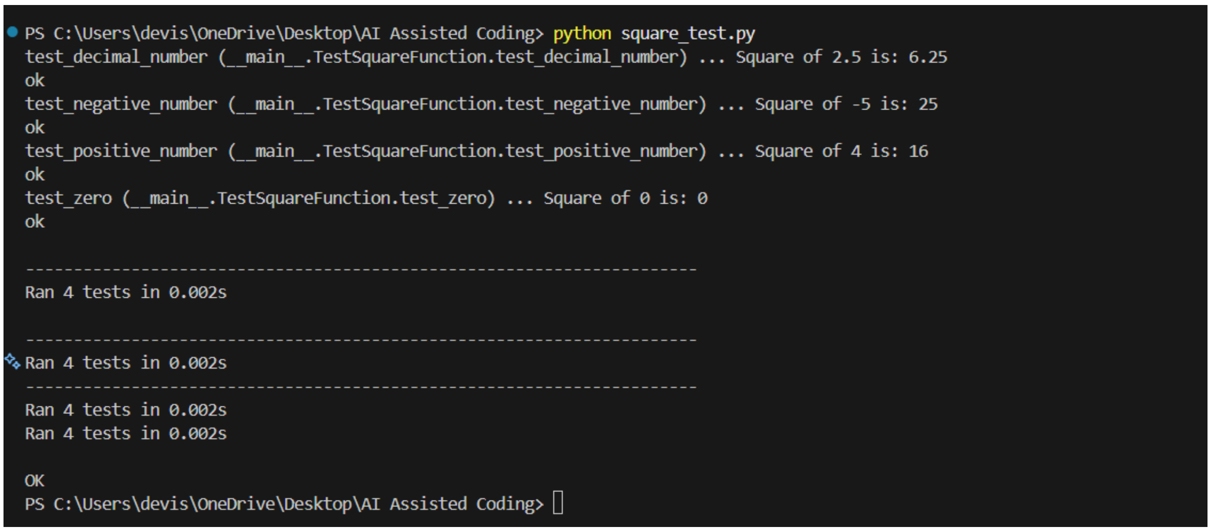


square_test.py ● square.py ×

square.py > square

```python
def square(x):
    return x * x
```

Output :



```
PS C:\Users\devis\OneDrive\Desktop\AI Assisted Coding> python square_test.py
test_decimal_number (__main__.TestSquareFunction.test_decimal_number) ... Square of 2.5 is: 6.25
ok
test_negative_number (__main__.TestSquareFunction.test_negative_number) ... Square of -5 is: 25
ok
test_positive_number (__main__.TestSquareFunction.test_positive_number) ... Square of 4 is: 16
ok
test_zero (__main__.TestSquareFunction.test_zero) ... Square of 0 is: 0
ok

----------------------------------------------------------------------
Ran 4 tests in 0.002s

----------------------------------------------------------------------
Ran 4 tests in 0.002s
----------------------------------------------------------------------
Ran 4 tests in 0.002s
Ran 4 tests in 0.002s

OK
PS C:\Users\devis\OneDrive\Desktop\AI Assisted Coding> 
```

Explanation :

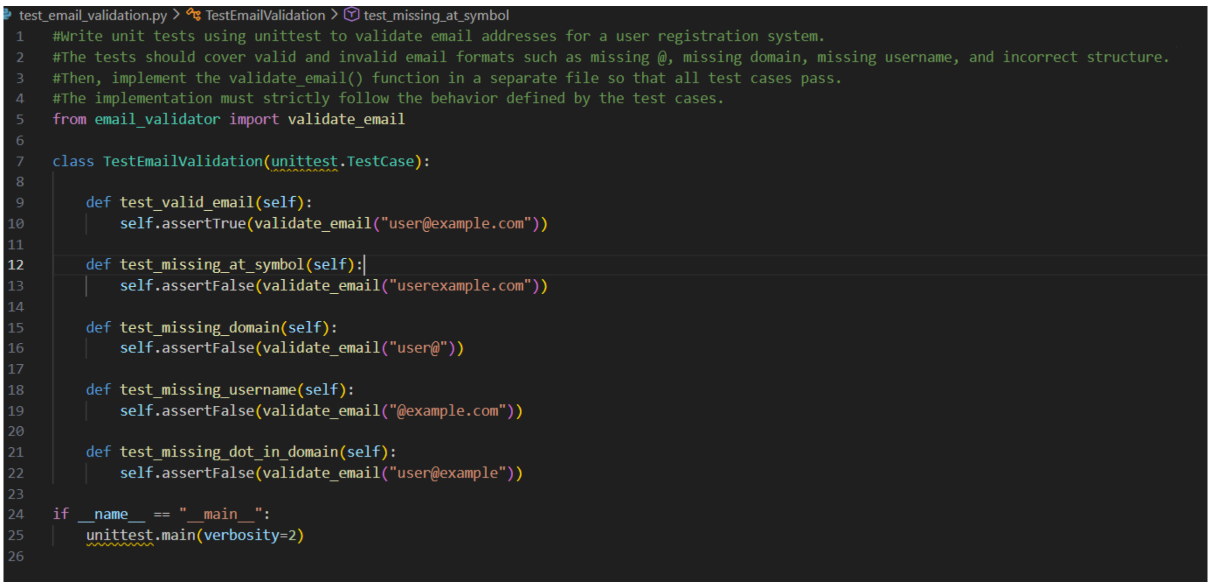
- We follow Test-Driven Development (TDD), where tests are written first.
- The test file (test_square.py) contains unit tests using the unit test module.
- These tests check the square(x) function for:
  - positive numbers
  - zero
  - negative numbers
  - decimal values
- Initially, tests fail because the function is not implemented.
- Then the function square(x) is implemented in a separate file (square.py).
- After implementation, all test cases pass successfully.
- This ensures the function works correctly for different inputs.

TASK – 02

Prompt : #Write unit tests using unittest to validate email addresses for a user registration system. The tests should cover valid and invalid email formats such as missing @, missing domain, missing username, and incorrect structure. Then, implement the validate_email() function in a separate file so that all test cases pass. The implementation must strictly follow the behavior defined by the test cases.

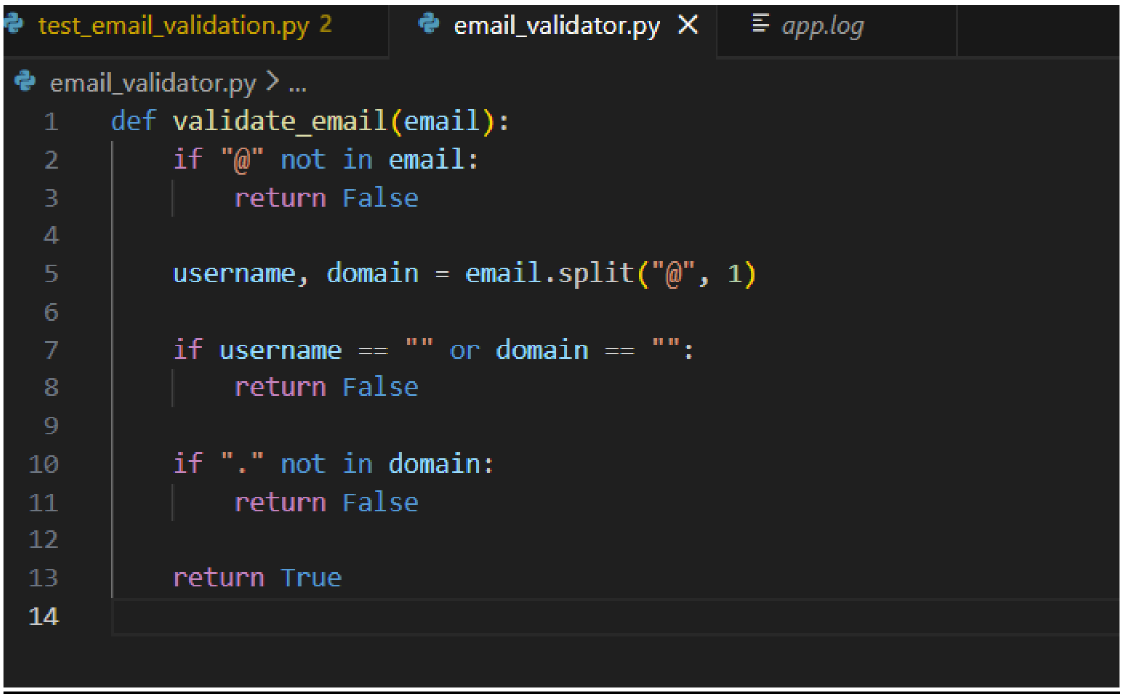
Code :

test_email_validator.py


test_email_validation.py > TestEmailValidation > test_missing_at_symbol

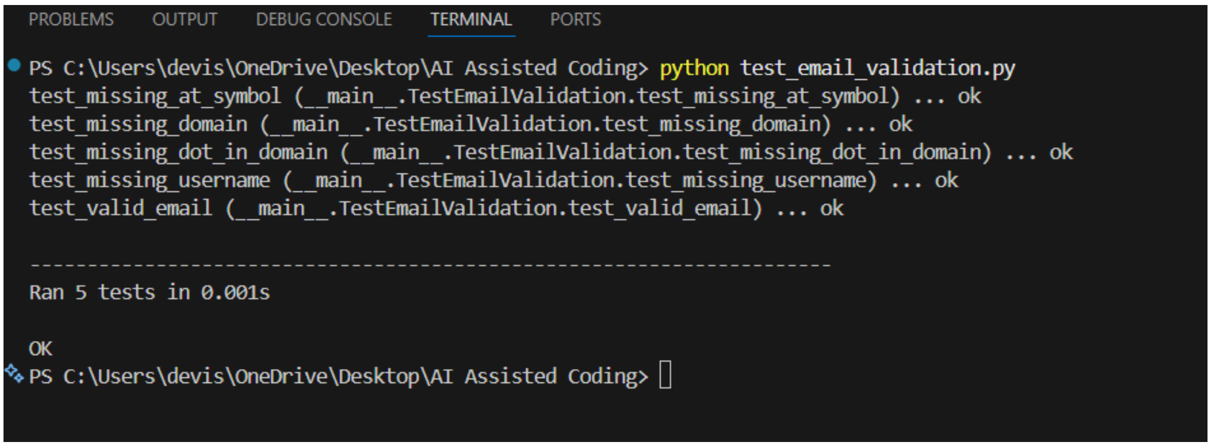
```python
#Write unit tests using unittest to validate email addresses for a user registration system.
#The tests should cover valid and invalid email formats such as missing @, missing domain, missing username, and incorrect structure.
#Then, implement the validate_email() function in a separate file so that all test cases pass.
#The implementation must strictly follow the behavior defined by the test cases.
from email_validator import validate_email

class TestEmailValidation(unittest.TestCase):

    def test_valid_email(self):
        self.assertTrue(validate_email("user@example.com"))

    def test_missing_at_symbol(self):
        self.assertFalse(validate_email("userexample.com"))

    def test_missing_domain(self):
        self.assertFalse(validate_email("user@"))

    def test_missing_username(self):
        self.assertFalse(validate_email("@example.com"))

    def test_missing_dot_in_domain(self):
        self.assertFalse(validate_email("user@example"))

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

email_validator.py



```python
def validate_email(email):
    if "@" not in email:
        return False

    username, domain = email.split("@", 1)

    if username == "" or domain == "":
        return False

    if "." not in domain:
        return False

    return True
```

Output :



```
PS C:\Users\devis\OneDrive\Desktop\AI Assisted Coding> python test_email_validation.py
test_missing_at_symbol (__main__.TestEmailValidation.test_missing_at_symbol) ... ok
test_missing_domain (__main__.TestEmailValidation.test_missing_domain) ... ok
test_missing_dot_in_domain (__main__.TestEmailValidation.test_missing_dot_in_domain) ... ok
test_missing_username (__main__.TestEmailValidation.test_missing_username) ... ok
test_valid_email (__main__.TestEmailValidation.test_valid_email) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
PS C:\Users\devis\OneDrive\Desktop\AI Assisted Coding>
```

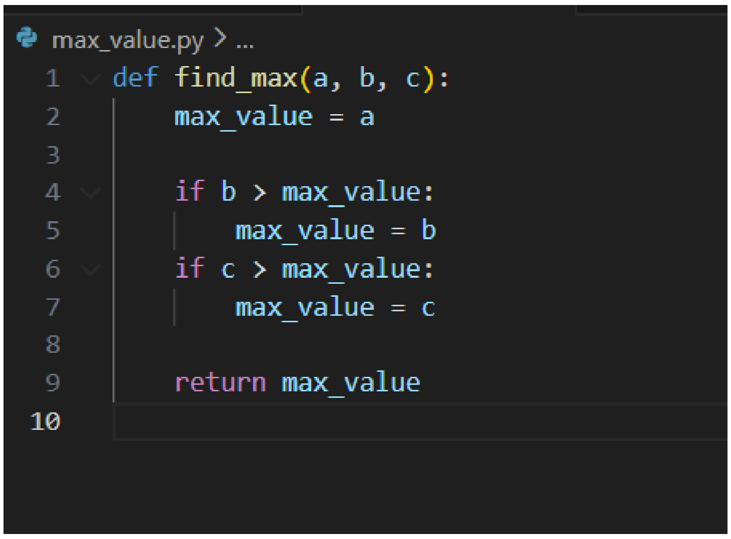
Explanation :

- Unit tests are written first to define valid and invalid email formats.
- Tests check cases like missing @, missing domain, and incorrect structure.
- The validate_email() function is implemented in a separate file.
- The function logic is written only to satisfy test expectations.
- This follows the Test-Driven Development (TDD) approach.

TASK – 03

Prompt : Write unit tests using unittest to determine the correct maximum value among three inputs. Test normal cases, equal values, negative numbers, and mixed values. After the tests are written, implement the find_max(a, b, c) function in a separate file so that all tests pass. The function logic must be derived strictly from the test cases.

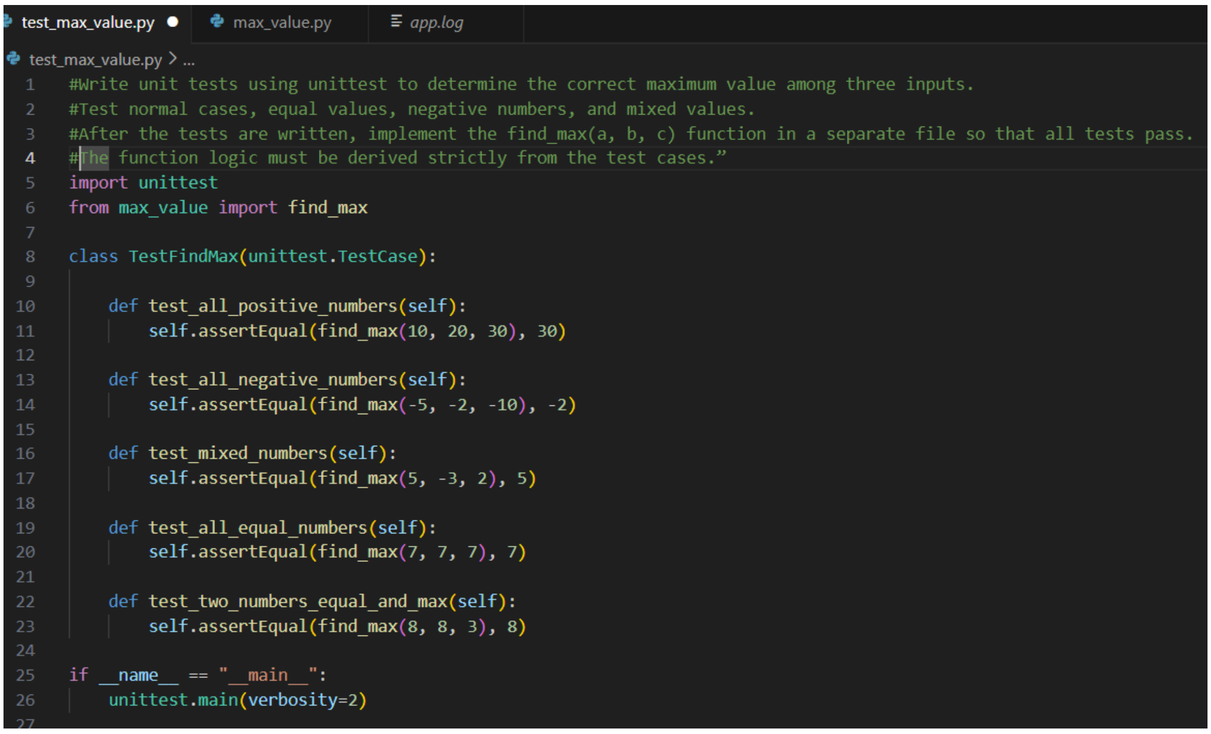
Code :



max_value.py > ...

```python
def find_max(a, b, c):
    max_value = a

    if b > max_value:
        max_value = b
    if c > max_value:
        max_value = c

    return max_value

```

test_max_value.py

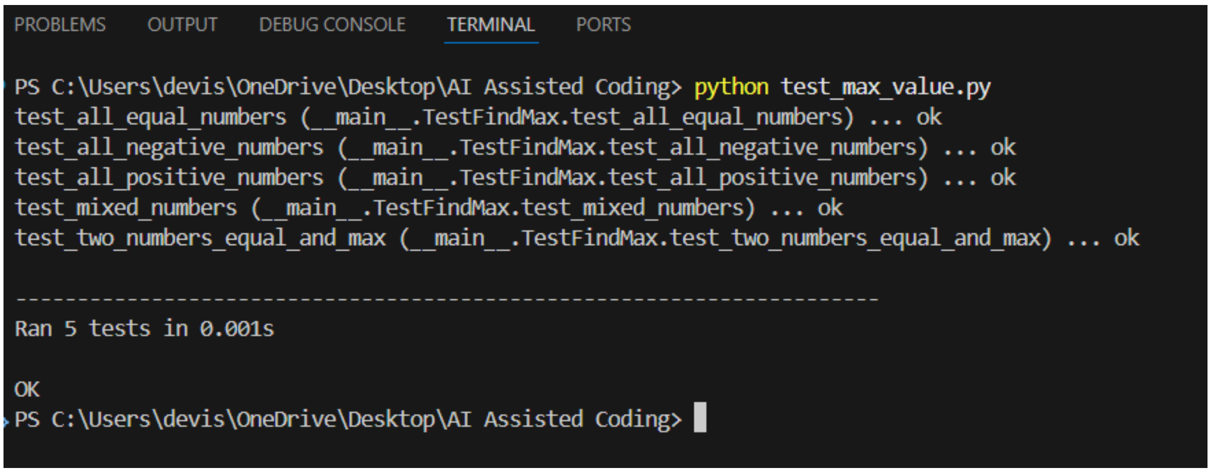


test_max_value.py ● | max_value.py | app.log

test_max_value.py > ...

```python
#Write unit tests using unittest to determine the correct maximum value among three inputs.
#Test normal cases, equal values, negative numbers, and mixed values.
#After the tests are written, implement the find_max(a, b, c) function in a separate file so that all tests pass.
#The function logic must be derived strictly from the test cases.”
import unittest
from max_value import find_max

class TestFindMax(unittest.TestCase):

    def test_all_positive_numbers(self):
        self.assertEqual(find_max(10, 20, 30), 30)

    def test_all_negative_numbers(self):
        self.assertEqual(find_max(-5, -2, -10), -2)

    def test_mixed_numbers(self):
        self.assertEqual(find_max(5, -3, 2), 5)

    def test_all_equal_numbers(self):
        self.assertEqual(find_max(7, 7, 7), 7)

    def test_two_numbers_equal_and_max(self):
        self.assertEqual(find_max(8, 8, 3), 8)

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

Output :



```
PS C:\Users\devis\OneDrive\Desktop\AI Assisted Coding> python test_max_value.py
test_all_equal_numbers (__main__.TestFindMax.test_all_equal_numbers) ... ok
test_all_negative_numbers (__main__.TestFindMax.test_all_negative_numbers) ... ok
test_all_positive_numbers (__main__.TestFindMax.test_all_positive_numbers) ... ok
test_mixed_numbers (__main__.TestFindMax.test_mixed_numbers) ... ok
test_two_numbers_equal_and_max (__main__.TestFindMax.test_two_numbers_equal_and_max) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
PS C:\Users\devis\OneDrive\Desktop\AI Assisted Coding>
```

Explanation :

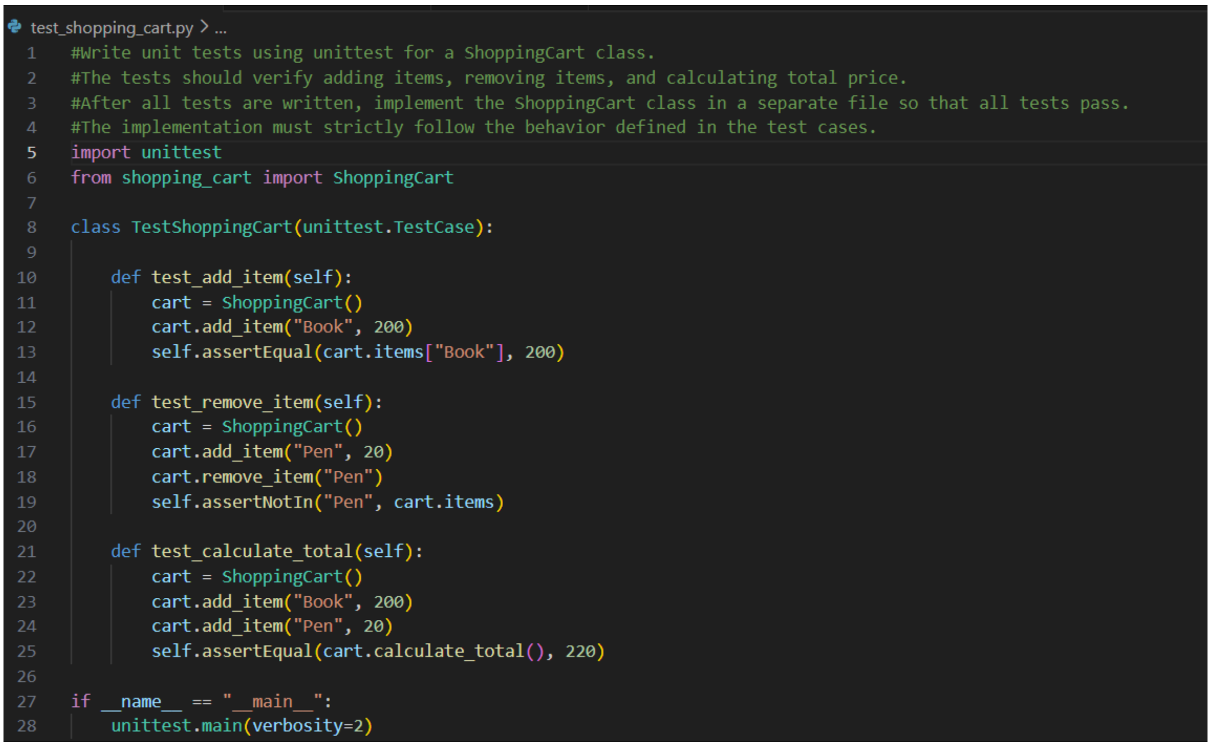
- Test cases are written first to define expected behavior.
- Tests cover positive numbers, negative numbers, equal values, and mixed cases.
- The function find_max() is implemented only after tests are completed.
- The logic is derived strictly from test expectations.
- This demonstrates the Test-Driven Development (TDD) approach.

TASK – 04

Prompt : Write unit tests using unittest for a ShoppingCart class.
The tests should verify adding items, removing items, and calculating total price. After all tests are written, implement the ShoppingCart class in a separate file so that all tests pass. The implementation must strictly follow the behavior defined in the test cases.

Code :

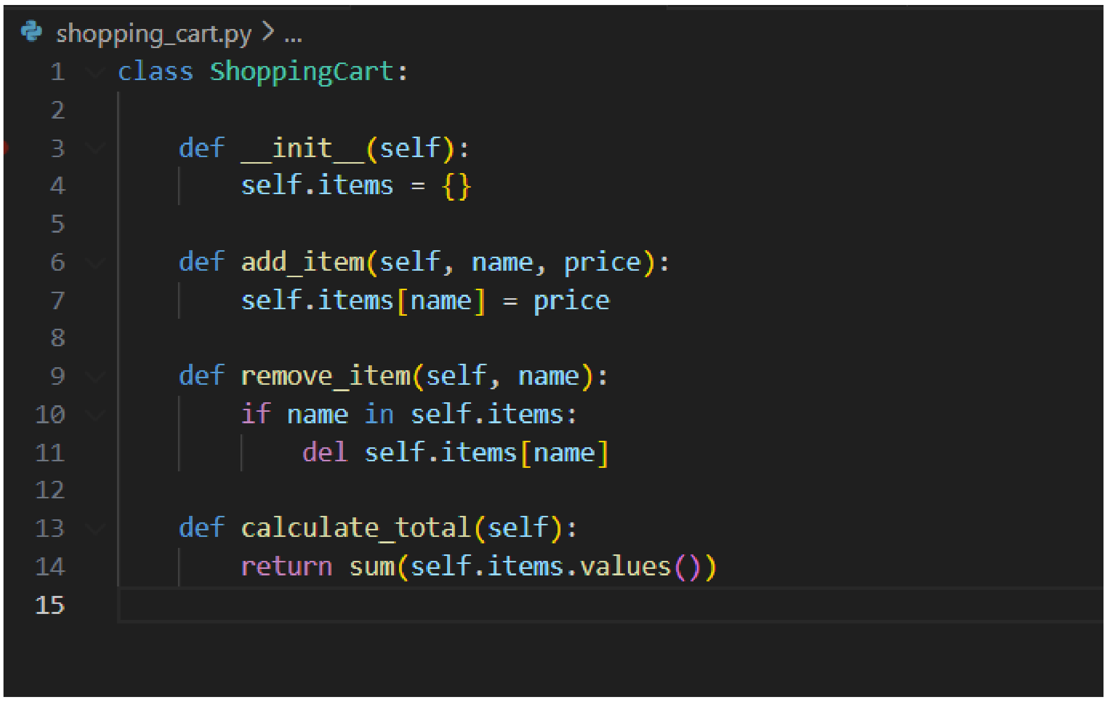
test_shopping_cart.py



test_shopping_cart.py > ...

```python
#Write unit tests using unittest for a ShoppingCart class.
#The tests should verify adding items, removing items, and calculating total price.
#After all tests are written, implement the ShoppingCart class in a separate file so that all tests pass.
#The implementation must strictly follow the behavior defined in the test cases.
import unittest
from shopping_cart import ShoppingCart

class TestShoppingCart(unittest.TestCase):

    def test_add_item(self):
        cart = ShoppingCart()
        cart.add_item("Book", 200)
        self.assertEqual(cart.items["Book"], 200)

    def test_remove_item(self):
        cart = ShoppingCart()
        cart.add_item("Pen", 20)
        cart.remove_item("Pen")
        self.assertNotIn("Pen", cart.items)

    def test_calculate_total(self):
        cart = ShoppingCart()
        cart.add_item("Book", 200)
        cart.add_item("Pen", 20)
        self.assertEqual(cart.calculate_total(), 220)

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

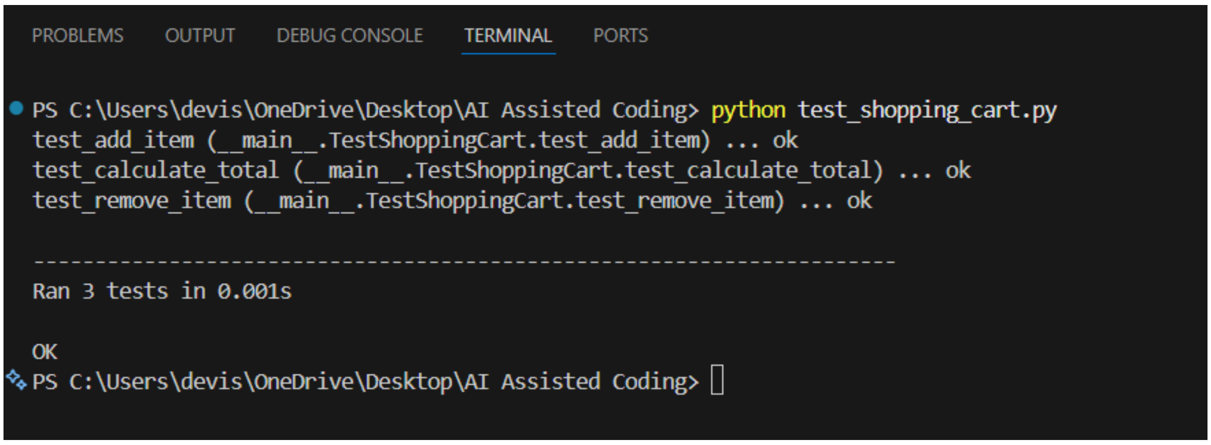
Shopping_cart.py



shopping_cart.py > ...

```python
class ShoppingCart:

    def __init__(self):
        self.items = {}

    def add_item(self, name, price):
        self.items[name] = price

    def remove_item(self, name):
        if name in self.items:
            del self.items[name]

    def calculate_total(self):
        return sum(self.items.values())

```

Output :



```
PS C:\Users\devis\OneDrive\Desktop\AI Assisted Coding> python test_shopping_cart.py
test_add_item (__main__.TestShoppingCart.test_add_item) ... ok
test_calculate_total (__main__.TestShoppingCart.test_calculate_total) ... ok
test_remove_item (__main__.TestShoppingCart.test_remove_item) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
PS C:\Users\devis\OneDrive\Desktop\AI Assisted Coding>
```

Explanation :

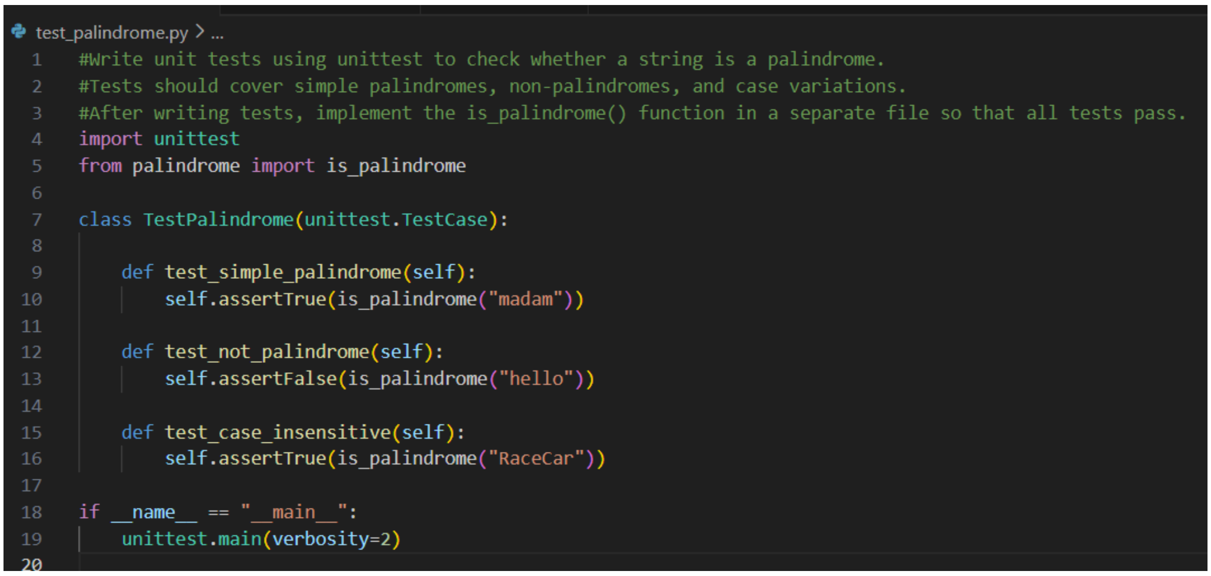
- Unit tests define expected cart behavior.
- Tests cover adding, removing items, and total calculation.
- ShoppingCart class is implemented after tests.
- All logic is derived from test expectations.
- This demonstrates TDD in a class-based design.

TASK – 05

Prompt : Write unit tests using unittest to check whether a string is a palindrome. Tests should cover simple palindromes, non-palindromes, and case variations. After writing tests, implement the is_palindrome() function in a separate file so that all tests pass.

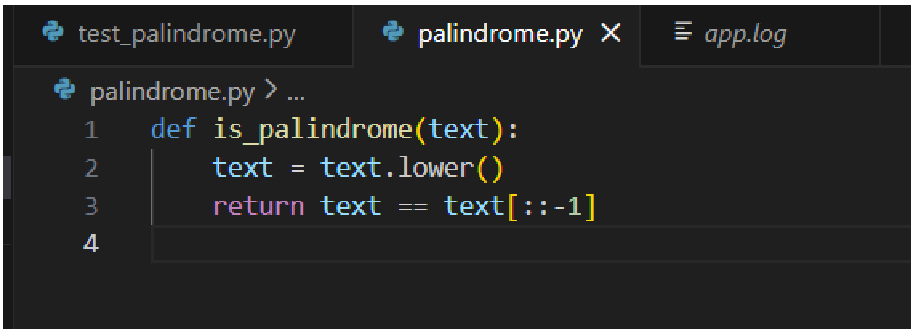
Code :

test_palindrome.py



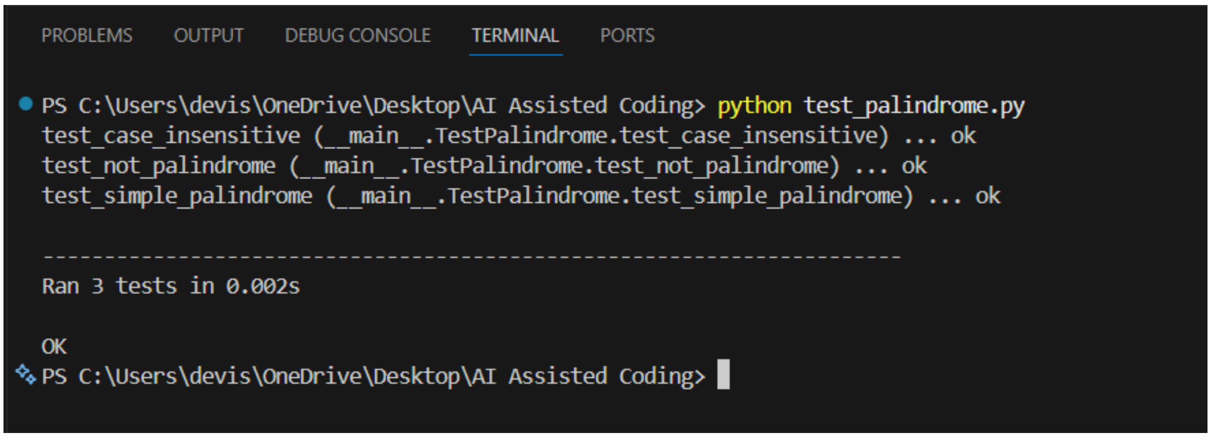
```python
#Write unit tests using unittest to check whether a string is a palindrome.
#Tests should cover simple palindromes, non-palindromes, and case variations.
#After writing tests, implement the is_palindrome() function in a separate file so that all tests pass.
import unittest
from palindrome import is_palindrome

class TestPalindrome(unittest.TestCase):

    def test_simple_palindrome(self):
        self.assertTrue(is_palindrome("madam"))

    def test_not_palindrome(self):
        self.assertFalse(is_palindrome("hello"))

    def test_case_insensitive(self):
        self.assertTrue(is_palindrome("RaceCar"))

if __name__ == "__main__":
    unittest.main(verbosity=2)
```

palindrome.py



```python
def is_palindrome(text):
    text = text.lower()
    return text == text[::-1]
```

Output :



```
PS C:\Users\devis\OneDrive\Desktop\AI Assisted Coding> python test_palindrome.py
test_case_insensitive (__main__.TestPalindrome.test_case_insensitive) ... ok
test_not_palindrome (__main__.TestPalindrome.test_not_palindrome) ... ok
test_simple_palindrome (__main__.TestPalindrome.test_simple_palindrome) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
PS C:\Users\devis\OneDrive\Desktop\AI Assisted Coding>
```

Explanation :

- Tests are written first to define palindrome behavior.
- Cases include valid, invalid, and case-insensitive strings.

- The function is implemented only after tests.
- All tests pass, proving correct TDD usage.