Assignment-8.4

2303A51304

Batch-05

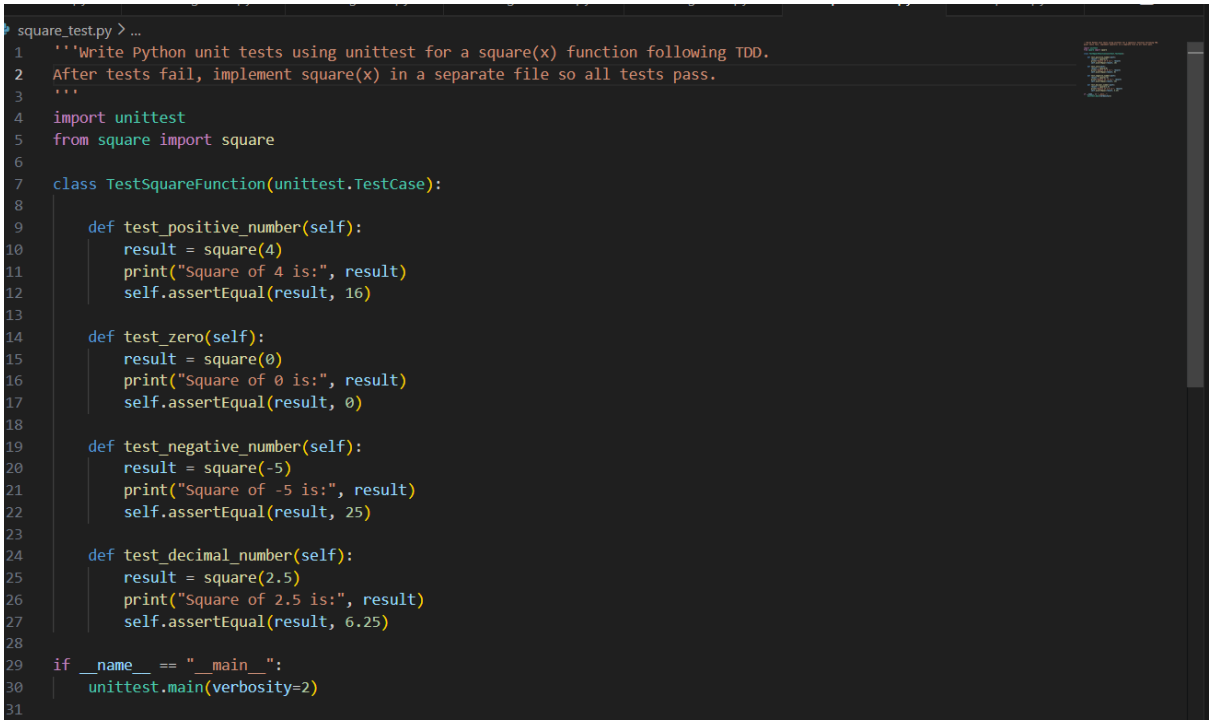
Task-01:

Prompt:

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.

Code:



square_test.py > ...

```
'''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)

```

```
b_without_fun.py | Asssign_3.3.py | Assign_5.4.py | Assignment-6.py | Assign-7.3.py | square_test.py | square.py
square.py > ...
1  def square(x):
2      return x * x
3
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\pooji\OneDrive\Desktop\AI_Assistance_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 0.001s

OK
```

Python
Python
Python
Python
Python
Python
Python
powershell
powershell
powershell
powershell

Explanation:

This program follows Test Driven Development (TDD) by writing unit tests before implementing the function.

The file square_test.py contains test cases to verify the square of positive, zero, negative, and decimal numbers.

Initially, the tests fail because the square() function is not implemented.

The function square(x) is then written in square.py to return x * x.

After implementation, all test cases pass successfully, confirming correct functionality.

Task-02:

Prompt:

Write unit tests first for an email validation function using TDD in Python,

then implement the function so all tests pass.

Code:

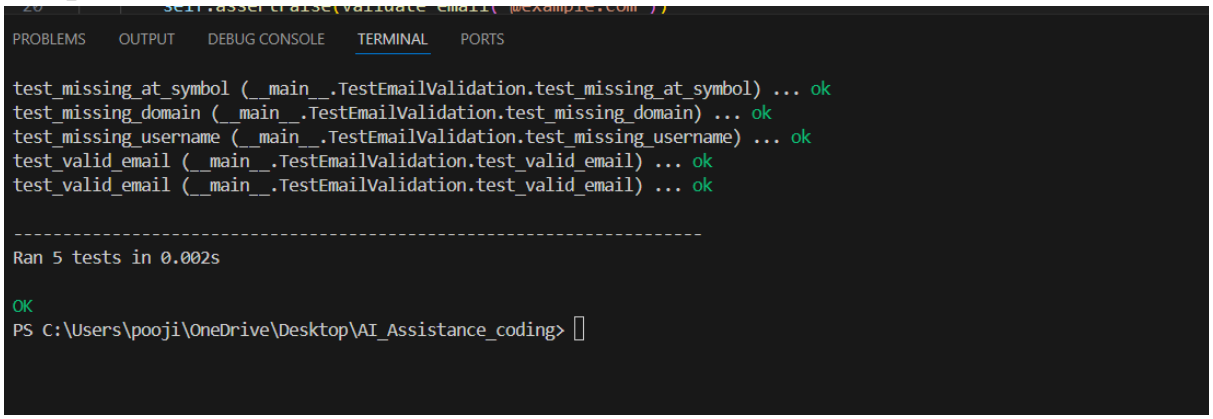
email_validator.py

```python

import re

def validate_email(email):
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, email))

```

test_email_validation.py

```python
'''Write unit tests first for an email validation function using TDD in Python,
then implement the function so all tests pass.'''


import unittest
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_invalid_structure(self):
        self.assertFalse(validate_email("user@com"))

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

```

Output:



Explanation:

Test Driven Development starts by writing unit tests before implementation.

The test file defines valid and invalid email formats using unittest.

Initially, tests fail because the validation function does not exist.

The validate_email() function is implemented to satisfy test conditions.

All tests pass, confirming correct email validation behavior.

Task-03:

Prompt:

Using Test Driven Development in Python, write unit tests first to verify a function max_of_three(a, b, c) that returns the maximum of three numbers.

Cover normal and edge cases. After tests are written, implement the function so all tests pass.

Code:

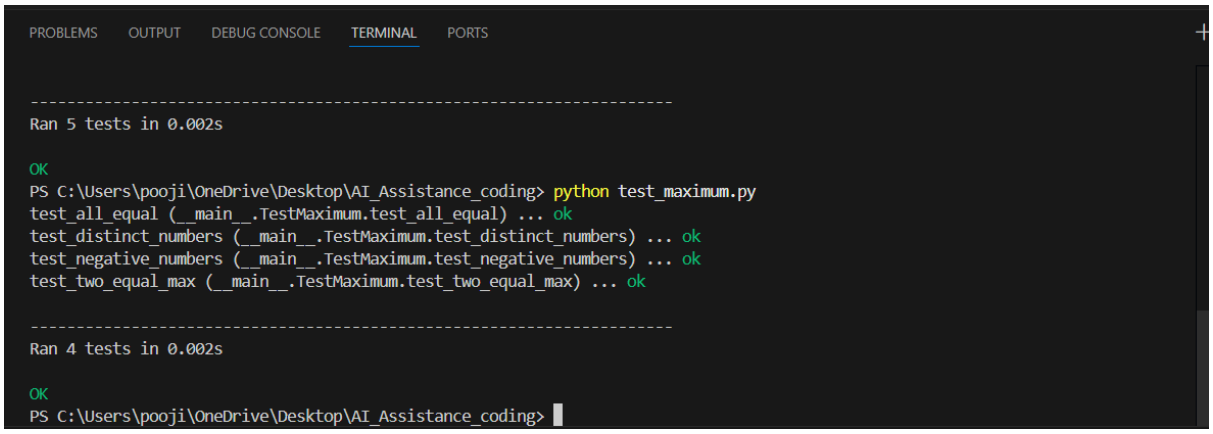
maximum.py > ...

```python

def max_of_three(a, b, c):
    return max(a, b, c)

```

test_maximum.py > ...

```python
import unittest
from maximum import max_of_three

class TestMaximum(unittest.TestCase):

    def test_distinct_numbers(self):
        self.assertEqual(max_of_three(3, 7, 5), 7)

    def test_all_equal(self):
        self.assertEqual(max_of_three(4, 4, 4), 4)

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

    def test_two_equal_max(self):
        self.assertEqual(max_of_three(6, 6, 2), 6)

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

```

Output:



Explanation:

Tests are written first to define expected behavior for maximum calculation. Different scenarios including edge cases are covered. Initially, tests fail due to missing implementation. The function is implemented strictly to satisfy test expectations. All tests passing confirms correct decision logic.

Task-04:

Prompt:

Follow TDD to build a ShoppingCart class in Python. First write unit tests for adding items, removing items, and calculating total price. After tests fail, implement the ShoppingCart class so all tests pass.

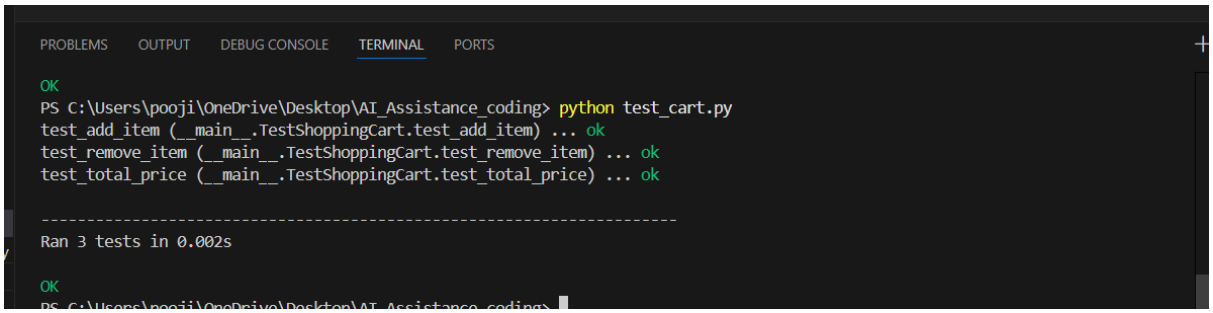
Code:

```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 total_price(self):
        return sum(self.items.values())
```

test_cart.py > TestShoppingCart > test_remove_item

```python
'''Follow TDD to build a ShoppingCart class in Python. First write unit tests for adding items, removing items,
and calculating total price.
After tests fail, implement the ShoppingCart class so all tests pass'''
import unittest
from cart import ShoppingCart

class TestShoppingCart(unittest.TestCase):

    def test_add_item(self):
        cart = ShoppingCart()
        cart.add_item("Book", 100)
        self.assertEqual(cart.total_price(), 100)

    def test_remove_item(self):
        cart = ShoppingCart()
        cart.add_item("Pen", 20)
        cart.remove_item("Pen")
        self.assertEqual(cart.total_price(), 0)

    def test_total_price(self):
        cart = ShoppingCart()
        cart.add_item("Bag", 500)
        cart.add_item("Shoes", 1500)
        self.assertEqual(cart.total_price(), 2000)

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

Output:



Explanation:

Unit tests define shopping cart behavior before implementation.

Tests validate add, remove, and total price operations.

The class is implemented only after tests are written. Logic is derived entirely from test expectations. Passing tests confirm correct cart functionality.

Task-05:

Prompt:

Using TDD in Python, write unit tests for an is_palindrome() function.

Cover simple palindromes, non-palindromes, and case variations.

After tests are complete, implement the function so all tests pass.

Code:

palindrome.py > ...

```python

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

```

test_palindrome.py > TestPalindrome

```python
'''Using TDD in Python, write unit tests for an is_palindrome() function.
Cover simple palindromes, non-palindromes, and case variations.
After tests are complete, implement the function so 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_non_palindrome(self):
        self.assertFalse(is_palindrome("hello"))

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

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

```

Output:

Explanation:

Tests define expected palindrome behavior before coding.

Different input cases ensure reliability.

The function is implemented after tests fail initially.

Logic matches test expectations exactly.

Passing tests confirm correct string validation.