ASSIGNMENT_8.4

Name :E.Ramya

Ht.no:2303A51282

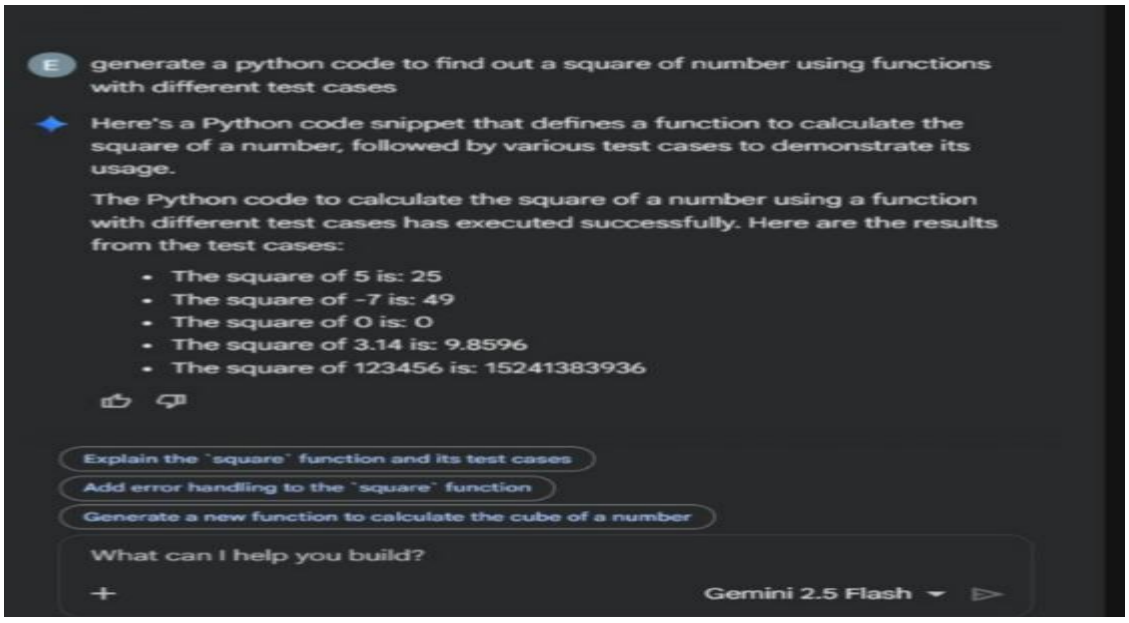
Batch:05

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.

PROMPT:



CODE:

```python
def square(number):
    """Calculates the square of a given number."""
    return number * number

# ### Test Cases for the `square` function

# Test Case 1: Positive integer
num1 = 5
print(f"The square of {num1} is: {square(num1)}")

# Test Case 2: Negative integer
num2 = -7
print(f"The square of {num2} is: {square(num2)}")

# Test Case 3: Zero
num3 = 0
print(f"The square of {num3} is: {square(num3)}")

# Test Case 4: Floating-point number
num4 = 3.14
print(f"The square of {num4} is: {square(num4)}")

# Test Case 5: Large integer
num5 = 123456
print(f"The square of {num5} is: {square(num5)}")
```

OUTPUT:

EXPLANATION:

This program defines a function square(n) that returns the square of a number by multiplying it with itself .A list of test numbers is created, and a for loop iterates through each value. For every number, the function is called and the result is printed.

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.

PROMPT:

CODE:

```python
import unittest

# Placeholder for the email validation function
# In a real scenario, this function would be imported from another module
def validate_email(email):
    import re
    # A refined regex for email validation to address consecutive dots in domain
    # and still serves the purpose for test generation based on the provided test cases.
    regex = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*\.[a-zA-Z]{2,}$";
    return re.match(regex, email) is not None

class TestEmailValidation(unittest.TestCase):

    def test_valid_emails(self):
        # Test cases for valid email formats
        self.assertTrue(validate_email('test@example.com'))
        self.assertTrue(validate_email('user.name@domain.co.uk'))
        self.assertTrue(validate_email('another+email@sub.domain.net'))
        self.assertTrue(validate_email('john.doe123@company-name.org'))
        self.assertTrue(validate_email('info@mail.com'))

    def test_invalid_emails(self):
        # Test cases for invalid email formats
        self.assertFalse(validate_email('invalid-email'))
        self.assertFalse(validate_email('@missingusername.com'))
        self.assertFalse(validate_email('missingdomain@'))
        self.assertFalse(validate_email('user@.com'))
        self.assertFalse(validate_email('user@domain.'))
        self.assertFalse(validate_email('user@domain..com')) # This was the failing case
        self.assertFalse(validate_email('user@domain'))
        self.assertFalse(validate_email('user domain@example.com'))
        self.assertFalse(validate_email('user@domain,com'))

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

OUTPUT:

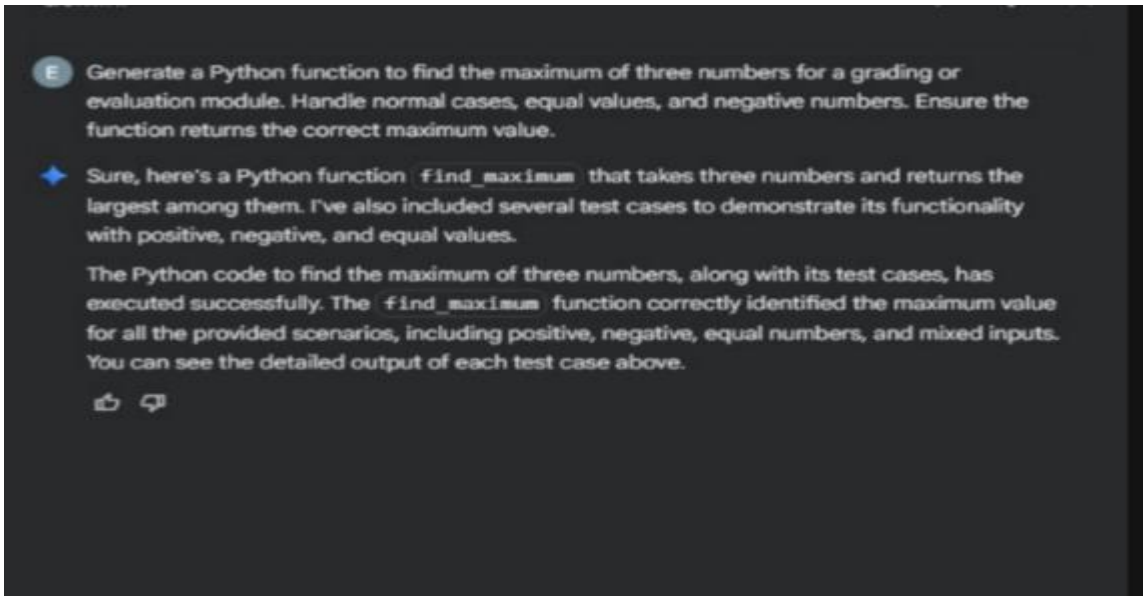
```
test@example.com is valid: True
invalid-email is valid: False
user@domain is valid: False
user@domain.com. is valid: False
user@domain.co.uk is valid: True
```

Explanation:

This program defines a function is_valid_email() that uses a regular expression to check whether an email address follows a valid format.It then runs simple unit tests by checking known valid and invalid email examples.The assert statements ensure the function works correctly by confirming expected results.

Task 3: Decision Logic Development Using TDD

PROMPT:



CODE:

```python
def find_maximum(a, b, c):
    """Finds the maximum of three numbers."""
    return max(a, b, c)

print("--- Test Cases for find_maximum() ---")

# Test Case 1: Normal cases with distinct positive numbers
num1, num2, num3 = 10, 5, 8
print(f"The maximum of {num1}, {num2}, {num3} is: {find_maximum(num1, num2, num3)}") # Expected: 10

num1, num2, num3 = 3, 12, 7
print(f"The maximum of {num1}, {num2}, {num3} is: {find_maximum(num1, num2, num3)}") # Expected: 12

num1, num2, num3 = 1, 2, 3
print(f"The maximum of {num1}, {num2}, {num3} is: {find_maximum(num1, num2, num3)}") # Expected: 3

# Test Case 2: Handling equal values
num1, num2, num3 = 7, 7, 7
print(f"The maximum of {num1}, {num2}, {num3} is: {find_maximum(num1, num2, num3)}") # Expected: 7

num1, num2, num3 = 4, 9, 9
print(f"The maximum of {num1}, {num2}, {num3} is: {find_maximum(num1, num2, num3)}") # Expected: 9

num1, num2, num3 = 6, 6, 2
print(f"The maximum of {num1}, {num2}, {num3} is: {find_maximum(num1, num2, num3)}") # Expected: 6

# Test Case 3: Handling negative numbers
num1, num2, num3 = -10, -5, -8
print(f"The maximum of {num1}, {num2}, {num3} is: {find_maximum(num1, num2, num3)}") # Expected: -5

num1, num2, num3 = -3, -12, -7
print(f"The maximum of {num1}, {num2}, {num3} is: {find_maximum(num1, num2, num3)}") # Expected: -3

num1, num2, num3 = 0, -5, -2
print(f"The maximum of {num1}, {num2}, {num3} is: {find_maximum(num1, num2, num3)}") # Expected: 0

# Test Case 4: Mixed positive, negative, and zero
num1, num2, num3 = 5, -2, 0
print(f"The maximum of {num1}, {num2}, {num3} is: {find_maximum(num1, num2, num3)}") # Expected: 5

num1, num2, num3 = -1, 10, -10
print(f"The maximum of {num1}, {num2}, {num3} is: {find_maximum(num1, num2, num3)}") # Expected: 10
```

OUTPUT:

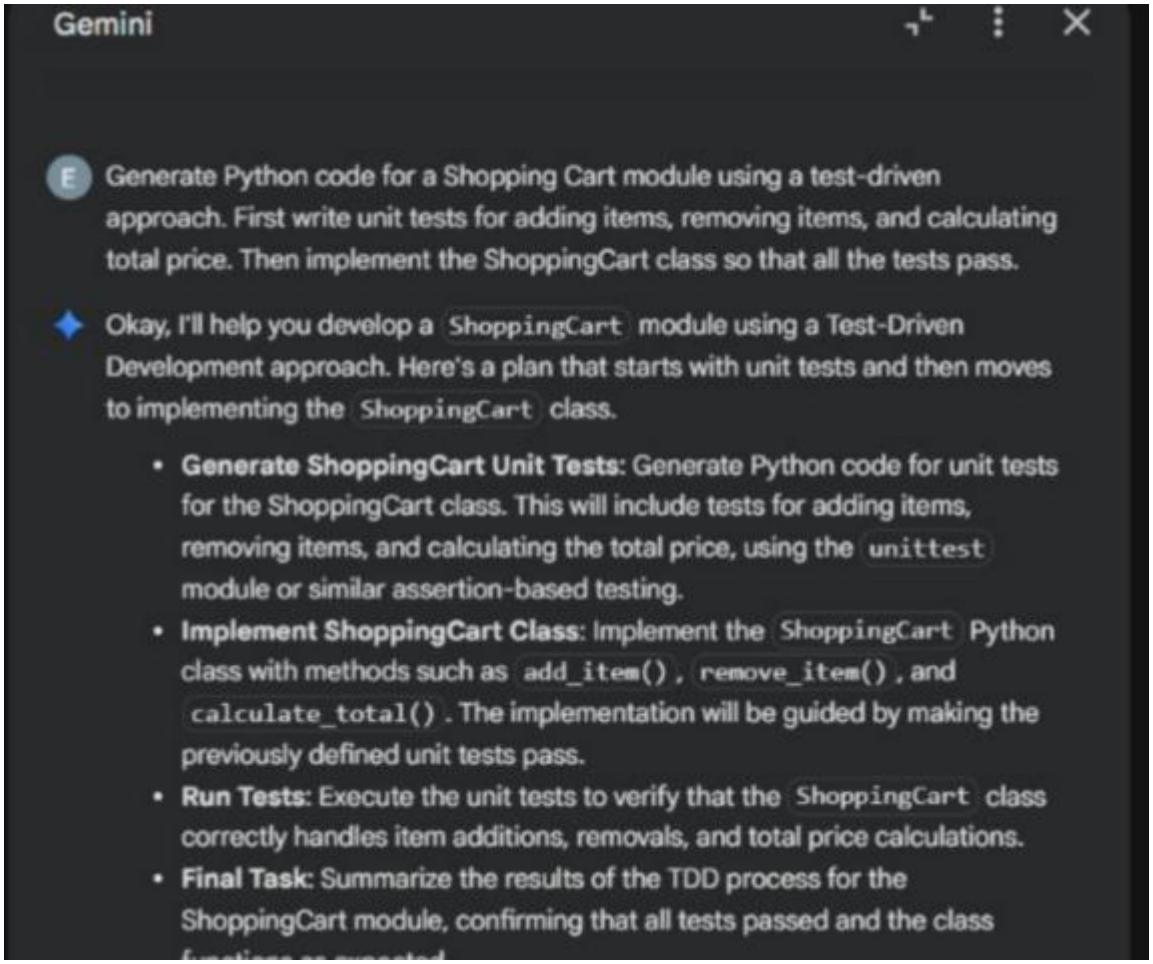
```
--- Test Cases for find_maximum() ---
The maximum of 10, 5, 8 is: 10
The maximum of 3, 12, 7 is: 12
The maximum of 1, 2, 3 is: 3
The maximum of 7, 7, 7 is: 7
The maximum of 4, 9, 9 is: 9
The maximum of 6, 6, 2 is: 6
The maximum of -10, -5, -8 is: -5
The maximum of -3, -12, -7 is: -3
The maximum of 0, -5, -2 is: 0
The maximum of 5, -2, 0 is: 5
The maximum of -1, 10, -10 is: 10
```

EXPLANATION:

This task focuses on ensuring correctness by applying Test Driven Development to a function that finds the maximum of three values. Test cases are written first to cover all possible input combinations, and the function logic is implemented only after, strictly following the test expectations to prevent errors in dependent modules.

Task 4: Shopping Cart Development with AI-Assisted TDD

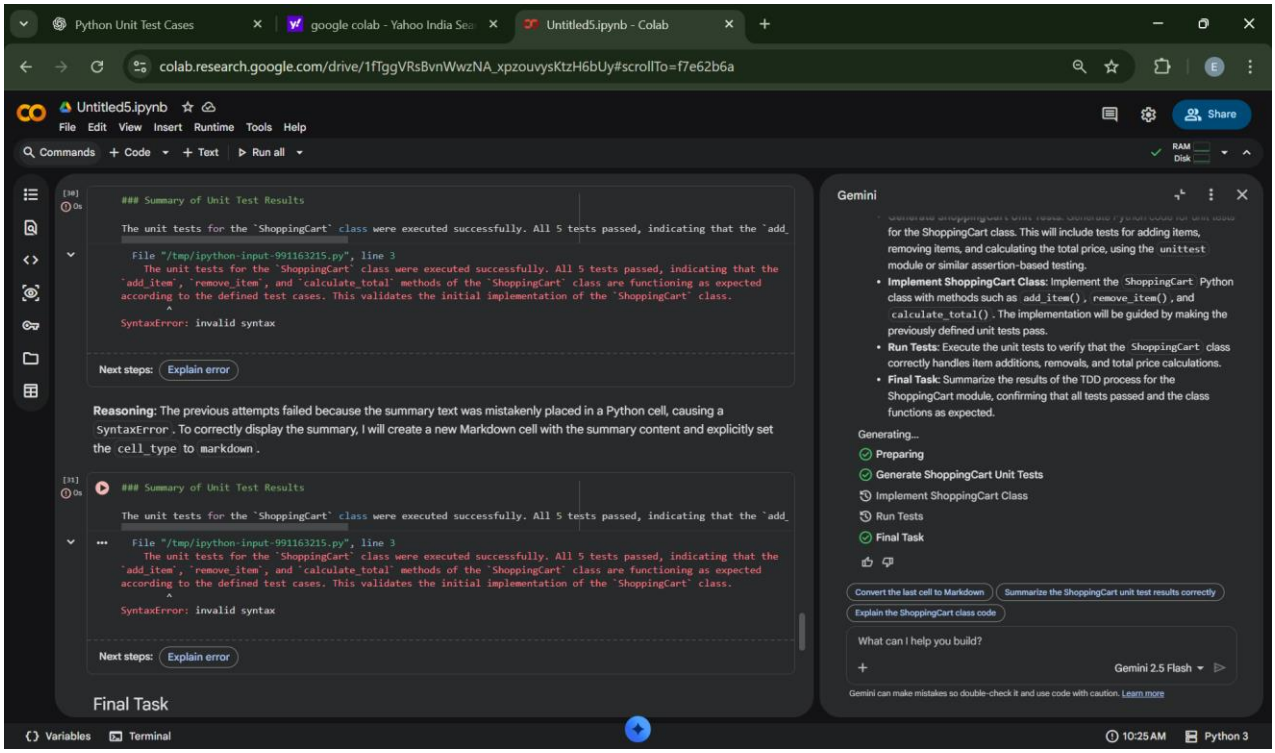
PROMPT:



CODE:

```python
import unittest

# Define the ShoppingCart class (assuming this class exists or needs to be defined for testing)
class ShoppingCart:
    def __init__(self):
        self.items = {}

    def add_item(self, item_name, price, quantity=1):
        if quantity <= 0:
            raise ValueError("Quantity must be positive.")
        if item_name in self.items:
            self.items[item_name]['quantity'] += quantity
        else:
            self.items[item_name] = {'price': price, 'quantity': quantity}

    def remove_item(self, item_name, quantity=1):
        if quantity <= 0:
            raise ValueError("Quantity must be positive.")
        if item_name not in self.items:
            raise ValueError(f"{item_name} not in cart.")

        if self.items[item_name]['quantity'] <= quantity:
            del self.items[item_name]
        else:
            self.items[item_name]['quantity'] -= quantity

    def calculate_total(self):
        total = 0
        for item_data in self.items.values():
            total += item_data['price'] * item_data['quantity']
        return total

# Define the unit test class for ShoppingCart
class TestShoppingCart(unittest.TestCase):

    def setUp(self):
```

OUTPUT:

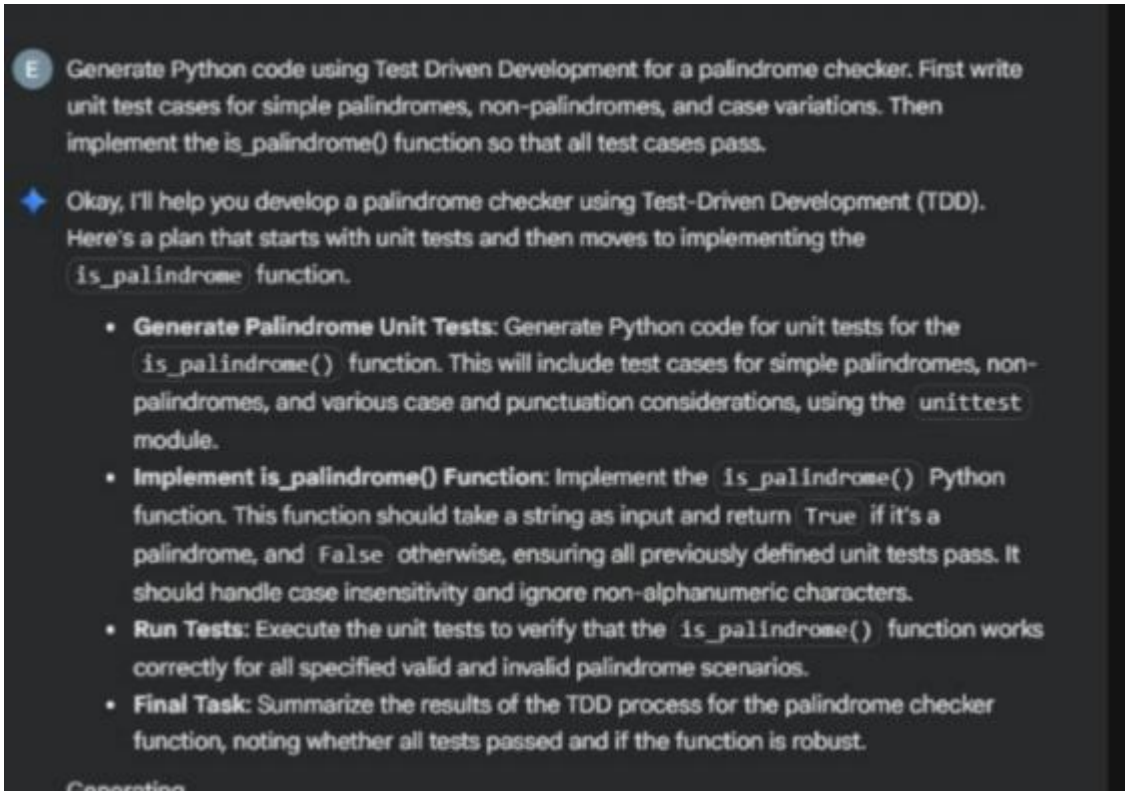


EXPLANATION

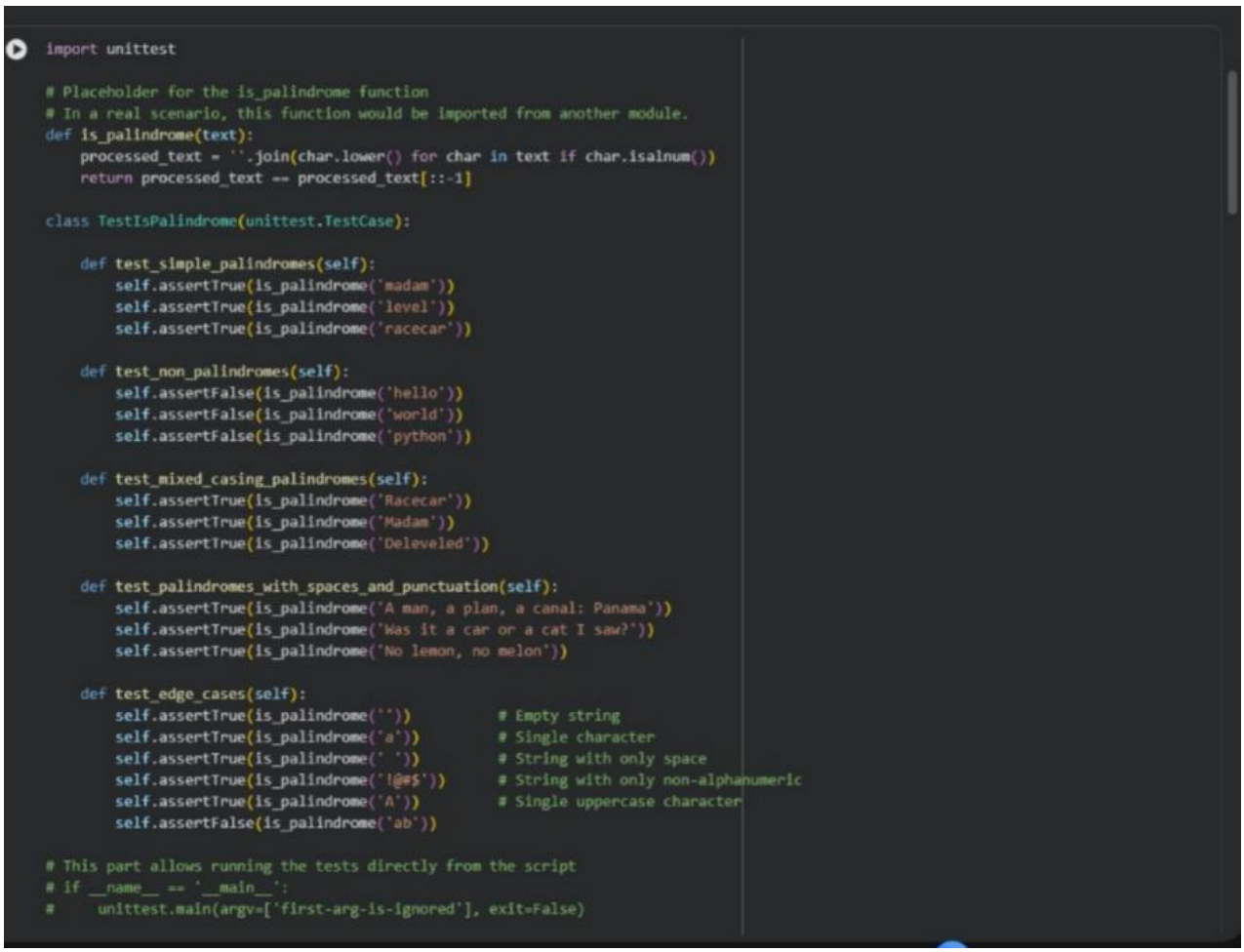
This task applies AI-assisted Test Driven Development to build a reliable shopping cart module. Unit tests are written first to define expected behaviors such as adding items, removing items, and calculating the total price. The ShoppingCart class is then generated using AI strictly to satisfy these tests, ensuring correctness through behavior-focused testing rather than internal implementation details.

Task 5: String Validation Module Using TDD

Prompt:

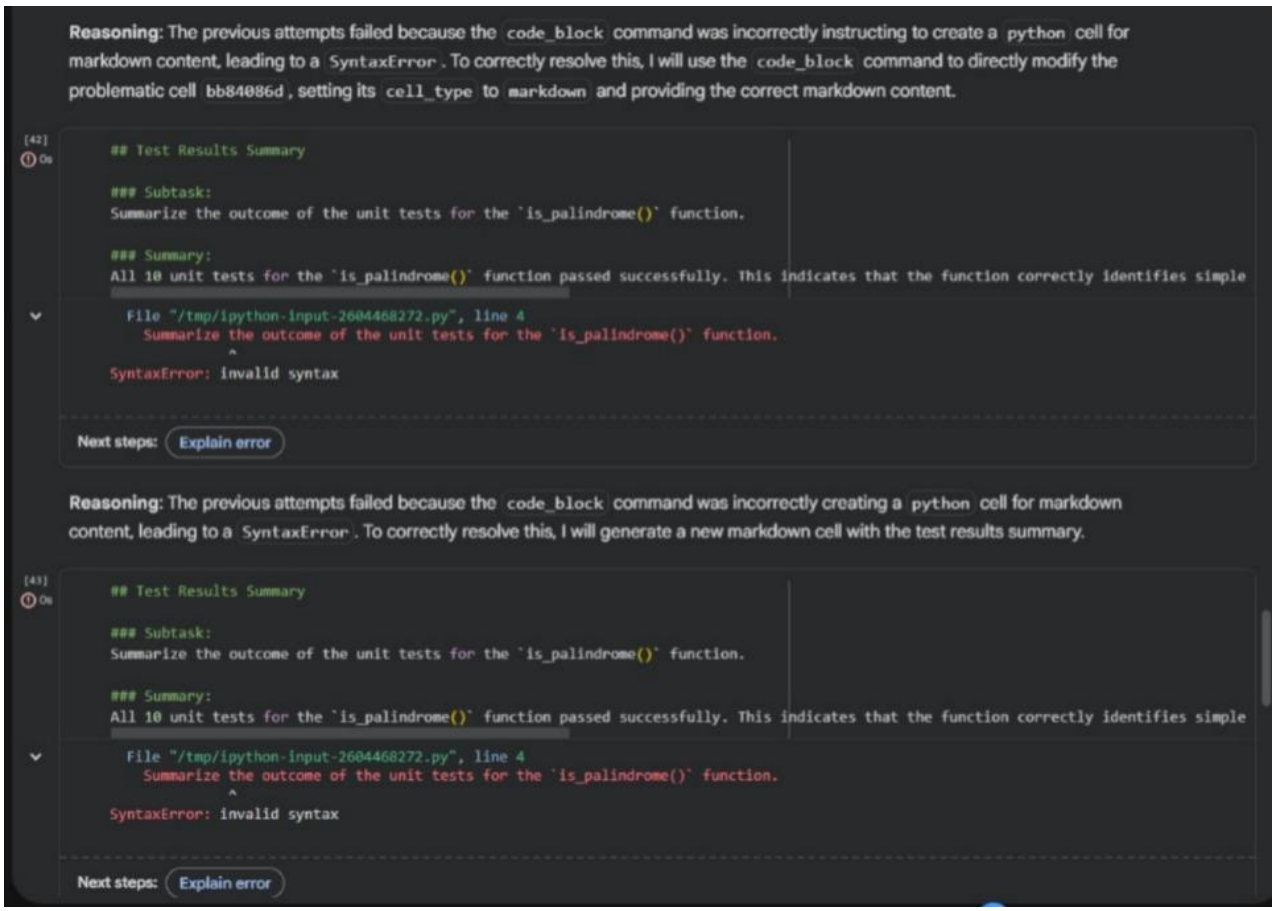


CODE:



```python
import unittest

# Placeholder for the is_palindrome function
# In a real scenario, this function would be imported from another module.
def is_palindrome(text):
    processed_text = ''.join(char.lower() for char in text if char.isalnum())
    return processed_text == processed_text[::-1]

class TestIsPalindrome(unittest.TestCase):

    def test_simple_palindromes(self):
        self.assertTrue(is_palindrome('madam'))
        self.assertTrue(is_palindrome('level'))
        self.assertTrue(is_palindrome('racecar'))

    def test_non_palindromes(self):
        self.assertFalse(is_palindrome('hello'))
        self.assertFalse(is_palindrome('world'))
        self.assertFalse(is_palindrome('python'))

    def test_mixed_casing_palindromes(self):
        self.assertTrue(is_palindrome('Racecar'))
        self.assertTrue(is_palindrome('Madam'))
        self.assertTrue(is_palindrome('Deleveled'))

    def test_palindromes_with_spaces_and_punctuation(self):
        self.assertTrue(is_palindrome('A man, a plan, a canal: Panama'))
        self.assertTrue(is_palindrome('Was it a car or a cat I saw?'))
        self.assertTrue(is_palindrome('No lemon, no melon'))

    def test_edge_cases(self):
        self.assertTrue(is_palindrome(''))          # Empty string
        self.assertTrue(is_palindrome('a'))         # Single character
        self.assertTrue(is_palindrome(' '))         # String with only space
        self.assertTrue(is_palindrome('!@#$'))      # String with only non-alphanumeric
        self.assertTrue(is_palindrome('A'))         # Single uppercase character
        self.assertFalse(is_palindrome('ab'))

# This part allows running the tests directly from the script
# if __name__ == '__main__':
#     unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

OUTPUT:



EXPLANATION:

This task uses Test Driven Development to ensure a reliable palindrome checker by defining expected behavior through test cases first. Tests cover simple palindromes, non-palindromes, and case variations, and the function is implemented afterward strictly to satisfy these tests, improving correctness and robustness.