

ASSIGNMENT_8.4

Name :E.Ramya

Ht.no:2303A51282

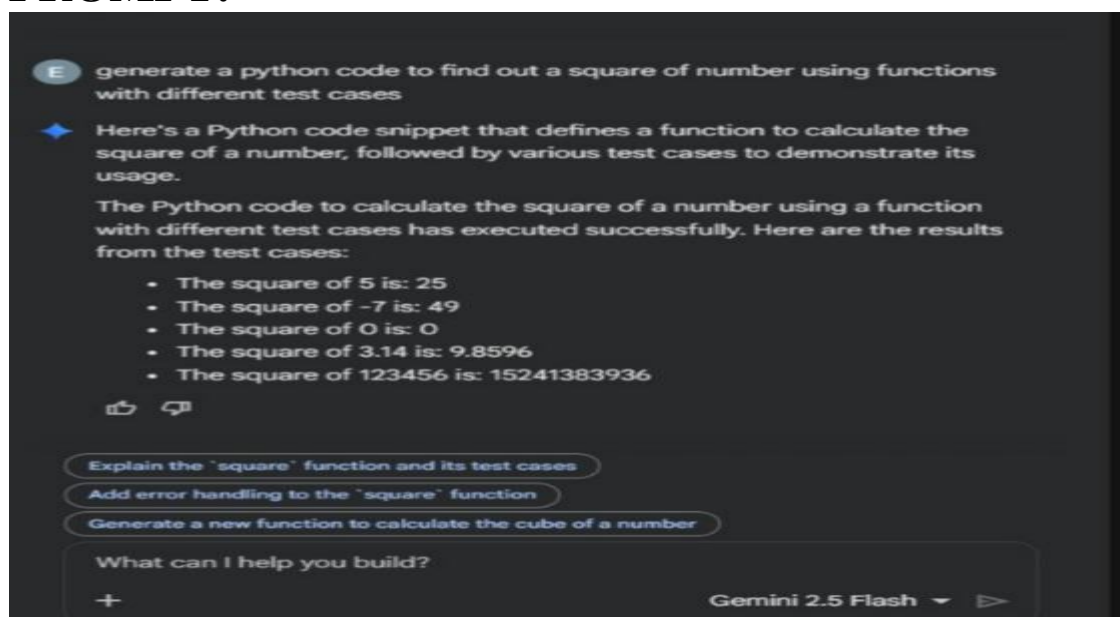
Batch:05

Task 1: Developing a Utility Function Using TDD

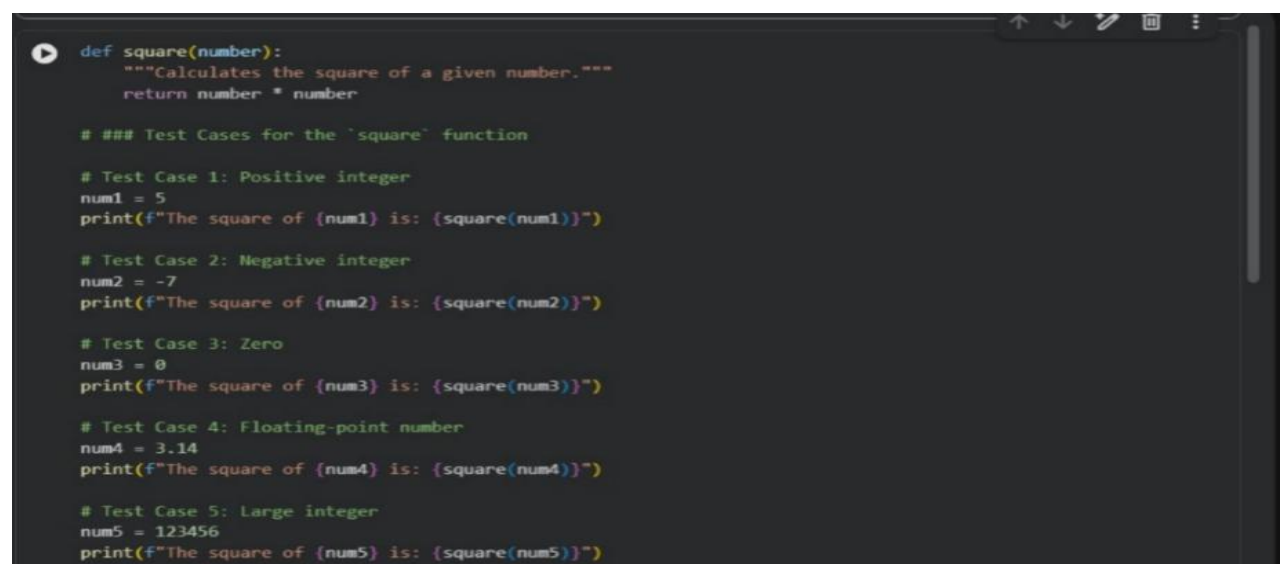
Scenario:

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:



OUTPUT:

```
*** The square of 5 is: 25
    The square of -7 is: 49
    The square of 0 is: 0
    The square of 3.14 is: 9.8596
    The square of 123456 is: 15241383936
```

EXPLANATION:

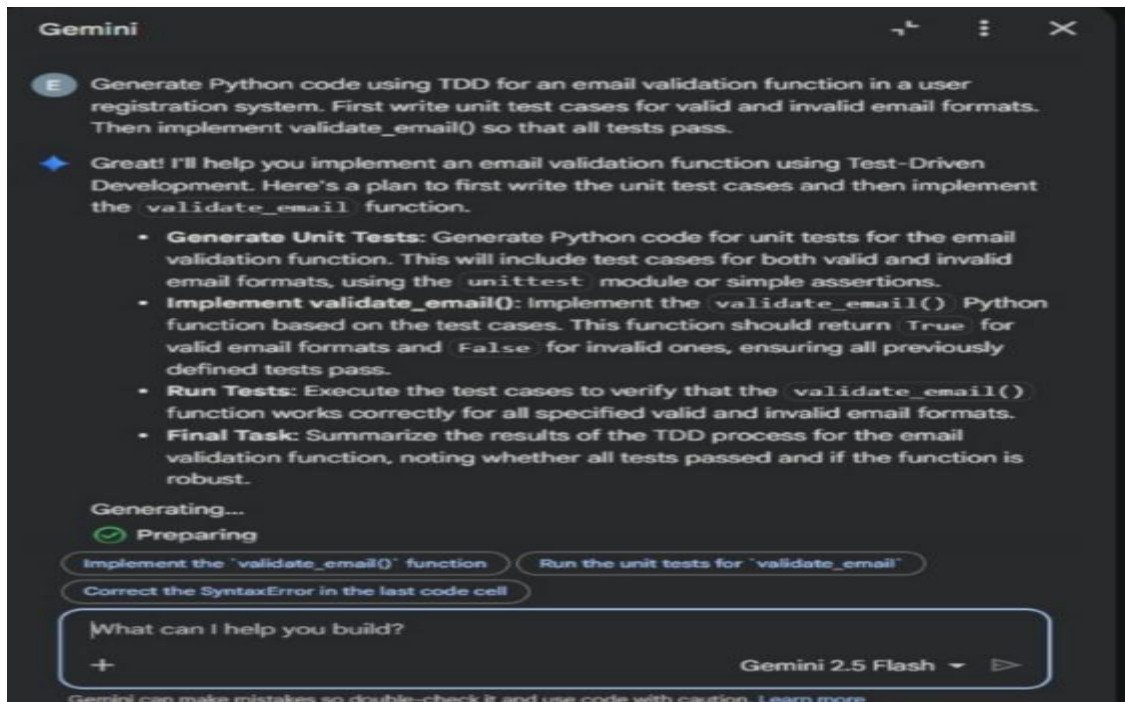
This program defines a function `square(n)` that returns the square of a number by multiplying it with itself. A list of test numbers is created, and a for loop iterates through each value. For every number, the function is called and the result is printed.

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:

```

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:

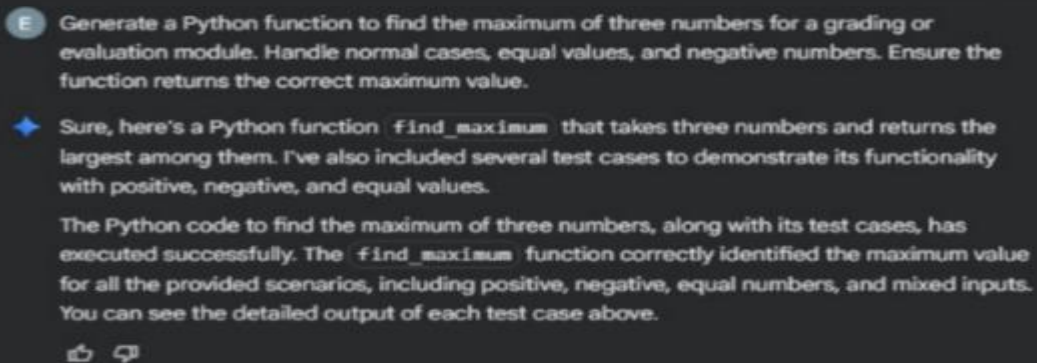
- test@example.com is valid: True
 invalid-email is valid: False
 user@domain is valid: False
user@domain.com. is valid: False
user@domain.co.uk is valid: True

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:



E Generate a Python function to find the maximum of three numbers for a grading or evaluation module. Handle normal cases, equal values, and negative numbers. Ensure the function returns the correct maximum value.

◆ Sure, here's a Python function `find_maximum` that takes three numbers and returns the largest among them. I've also included several test cases to demonstrate its functionality with positive, negative, and equal values.

The Python code to find the maximum of three numbers, along with its test cases, has executed successfully. The `find_maximum` function correctly identified the maximum value for all the provided scenarios, including positive, negative, equal numbers, and mixed inputs. You can see the detailed output of each test case above.

👍 🗨

CODE:

```
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
```

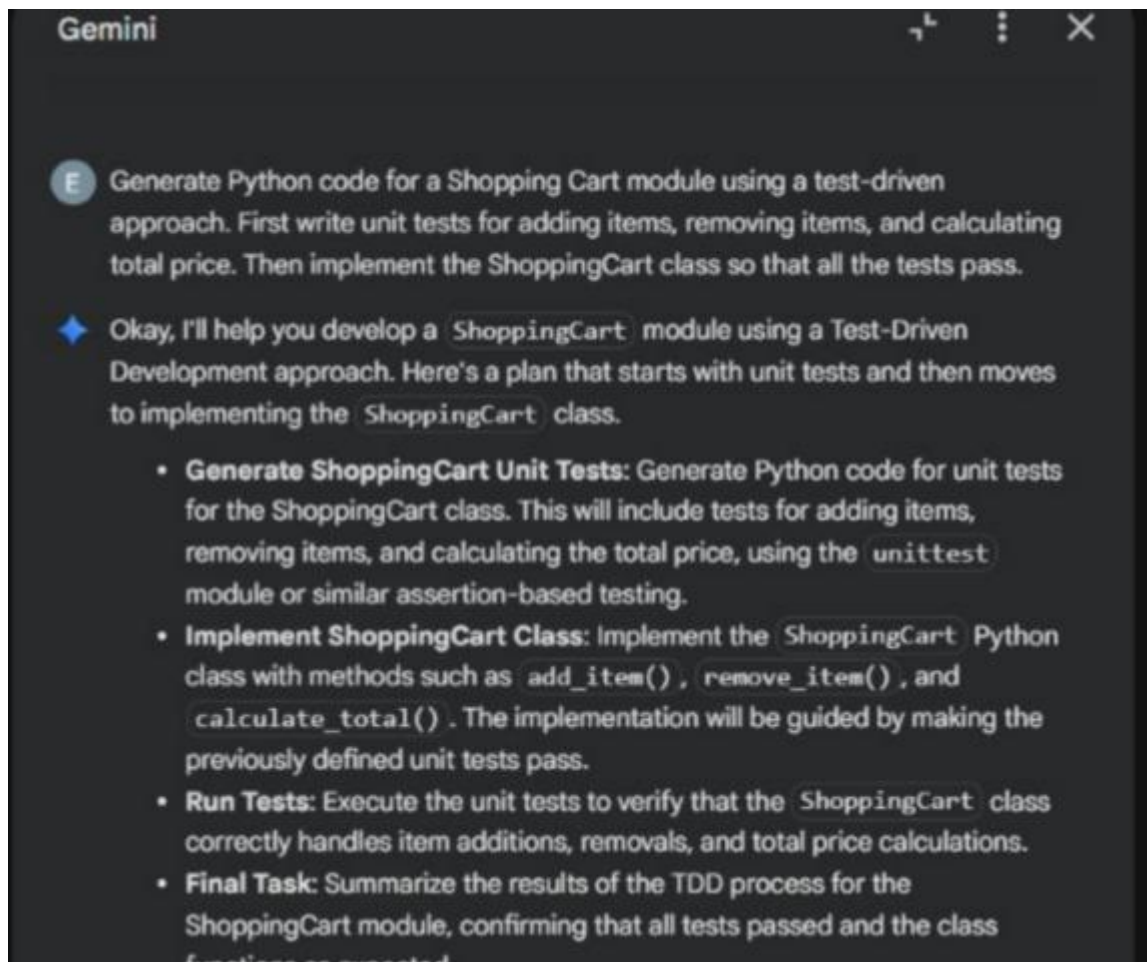
EXPLANATION:

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:

```
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.")

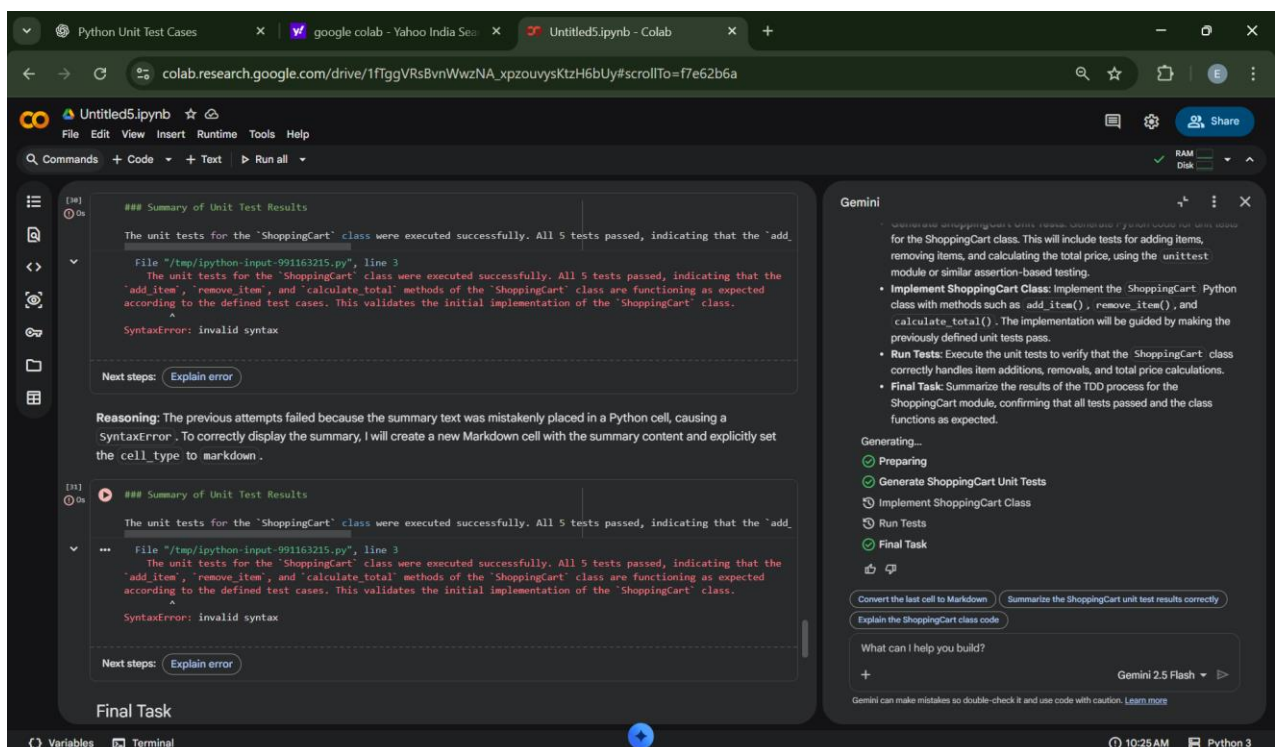
        if self.items[item_name]['quantity'] <= quantity:
            del self.items[item_name]
        else:
            self.items[item_name]['quantity'] -= quantity

    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:



Python Unit Test Cases

colab.research.google.com/drive/1fTggVRsBvnWwzNA_xpzouyvsKtzH6bUy#scrollTo=f7e62b6a

Untitled5.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

Summary of Unit Test Results

The unit tests for the 'ShoppingCart' class were executed successfully. All 5 tests passed, indicating that the 'add_item', 'remove_item', and 'calculate_total' methods of the 'ShoppingCart' class are functioning as expected according to the defined test cases. This validates the initial implementation of the 'ShoppingCart' class.

SyntaxError: invalid syntax

Next steps: Explain error

Reasoning: The previous attempts failed because the summary text was mistakenly placed in a Python cell, causing a SyntaxError. To correctly display the summary, I will create a new Markdown cell with the summary content and explicitly set the cell_type to markdown.

Summary of Unit Test Results

The unit tests for the 'ShoppingCart' class were executed successfully. All 5 tests passed, indicating that the 'add_item', 'remove_item', and 'calculate_total' methods of the 'ShoppingCart' class are functioning as expected according to the defined test cases. This validates the initial implementation of the 'ShoppingCart' class.

SyntaxError: invalid syntax

Next steps: Explain error

Final Task

Gemini

- Implement ShoppingCart Class: Implement the 'ShoppingCart' Python class with methods such as add_item(), remove_item(), and calculate_total(). The implementation will be guided by making the previously defined unit tests pass.
- Run Tests: Execute the unit tests to verify that the 'ShoppingCart' class correctly handles item additions, removals, and total price calculations.
- Final Task: Summarize the results of the TDD process for the ShoppingCart module, confirming that all tests passed and the class functions as expected.

Generating...

Preparing

Generate ShoppingCart Unit Tests

Implement ShoppingCart Class

Run Tests

Final Task

Convert the last cell to Markdown Summarize the ShoppingCart unit test results correctly

Explain the ShoppingCart class code

What can I help you build?

Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. Learn more

Variables Terminal

10:25 AM Python 3

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:



The screenshot shows a chat interface with a dark background. On the left, there is a vertical sidebar with a blue circle containing a white 'E' and a blue diamond icon. The main area contains the following text:

E Generate Python code using Test Driven Development for a palindrome checker. First write unit test cases for simple palindromes, non-palindromes, and case variations. Then implement the `is_palindrome()` function so that all test cases pass.

◆ Okay, I'll help you develop a palindrome checker using Test-Driven Development (TDD). Here's a plan that starts with unit tests and then moves to implementing the `is_palindrome` function.

- **Generate Palindrome Unit Tests:** Generate Python code for unit tests for the `is_palindrome()` function. This will include test cases for simple palindromes, non-palindromes, and various case and punctuation considerations, using the `unittest` module.
- **Implement `is_palindrome()` Function:** Implement the `is_palindrome()` Python function. This function should take a string as input and return `True` if it's a palindrome, and `False` otherwise, ensuring all previously defined unit tests pass. It should handle case insensitivity and ignore non-alphanumeric characters.
- **Run Tests:** Execute the unit tests to verify that the `is_palindrome()` function works correctly for all specified valid and invalid palindrome scenarios.
- **Final Task:** Summarize the results of the TDD process for the palindrome checker function, noting whether all tests passed and if the function is robust.

Generation

CODE:

```
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:

Reasoning: The previous attempts failed because the `code_block` command was incorrectly instructing to create a `python` cell for markdown content, leading to a `SyntaxError`. To correctly resolve this, I will use the `code_block` command to directly modify the problematic cell `bb84086d`, setting its `cell_type` to `markdown` and providing the correct markdown content.

```
[42]
① Os
## Test Results Summary

### Subtask:
Summarize the outcome of the unit tests for the 'is_palindrome()' function.

### Summary:
All 10 unit tests for the 'is_palindrome()' function passed successfully. This indicates that the function correctly identifies simple
File "/tmp/ipython-input-2604468272.py", line 4
    Summarize the outcome of the unit tests for the 'is_palindrome()' function.
    ^
SyntaxError: invalid syntax
```

Next steps: [Explain error](#)

Reasoning: The previous attempts failed because the `code_block` command was incorrectly creating a `python` cell for markdown content, leading to a `SyntaxError`. To correctly resolve this, I will generate a new markdown cell with the test results summary.

```
[43]
① Os
## Test Results Summary

### Subtask:
Summarize the outcome of the unit tests for the 'is_palindrome()' function.

### Summary:
All 10 unit tests for the 'is_palindrome()' function passed successfully. This indicates that the function correctly identifies simple
File "/tmp/ipython-input-2604468272.py", line 4
    Summarize the outcome of the unit tests for the 'is_palindrome()' function.
    ^
SyntaxError: invalid syntax
```

Next steps: [Explain error](#)

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.