

AI ASSISTANT CODING

ASSIGNMENT-8.4

Name: T. Swetha

Roll No:2303a51317

Batch:05

Task 1: Developing a Utility Function Using TDD

Prompt:

- T** Give me code for finding the square of a given using differnt test cases
- ✦ Certainly! Here's the code to calculate the square of numbers using different test cases. This will define a `square` function and then apply it to each number in the `test_numbers` list.

Code:

```
[3]
✓ 0s
# function to calculate square
def square(n):
    return n * n

# Define multiple test cases
test_numbers = [3, 8, 1, -5, 0, 12]

# Iterate through the test cases and print the square of each number
for num in test_numbers:
    print(f"Square of {num} is {square(num)}")
```

Output:

```
print(f"Square of {num} is {square(num)}")

... Square of 3 is 9
      Square of 8 is 64
      Square of 1 is 1
      Square of -5 is 25
      Square of 0 is 0
      Square of 12 is 144
```

Code 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

Prompt:

- give code for Email Validation for a User Registration System include Writing unit test cases that define valid and invalid email formats (e.g., missing @, missing domain, incorrect structure).
- Certainly! Here's Python code for email validation, complete with unit test cases to check for valid and invalid email formats. This solution uses regular expressions for robust validation.

Code:

```
[Z]
✓ 0s
import re

def is_valid_email(email):
    """
    Validates an email address against a common regular expression pattern.
    This pattern is a general one and may not cover all edge cases defined by RFCs,
    but it's widely used for practical email validation in registration systems.
    """
    # A fairly comprehensive regex for email validation
    # Modified to disallow consecutive dots in both local part and domain part
    email_regex = r"^[a-zA-Z0-9_%+]+(?:\.[a-zA-Z0-9_%+]+)*@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*\.[a-zA-Z0-9-]+$"
    if re.fullmatch(email_regex, email):
        return True
    else:
        return False

print("Email validation function defined.")

... Email validation function defined.
```

```
[9] 0s  print("Running simplified unit tests for email validation:")

# --- Simplified Unit Test Cases ---

simplified_valid_emails = [
    "test@example.com",
    "user123@mail.net"
]

simplified_invalid_emails = [
    "missingat.com",      # Missing @ symbol
    "user@domain"        # Missing top-level domain
]

# Test Valid Emails
print("\n--- Testing Valid Emails (Simplified) ---")
for email in simplified_valid_emails:
    result = is_valid_email(email)
    print(f"'{email}' is valid: {result} (Expected: True)")
    assert result is True, f"Expected '{email}' to be valid, but it was marked invalid."

# Test Invalid Emails
print("\n--- Testing Invalid Emails (Simplified) ---")
for email in simplified_invalid_emails:
    result = is_valid_email(email)
    print(f"'{email}' is valid: {result} (Expected: False)")
    assert result is False, f"Expected '{email}' to be invalid, but it was marked valid."

print("\nSimplified unit tests passed successfully!")
```

Output:

```
... Running simplified unit tests for email validation:

--- Testing Valid Emails (Simplified) ---
'test@example.com' is valid: True (Expected: True)
'user123@mail.net' is valid: True (Expected: True)

--- Testing Invalid Emails (Simplified) ---
'missingat.com' is valid: False (Expected: False)
'user@domain' is valid: False (Expected: False)

Simplified unit tests passed successfully!
```

Code Explanation:

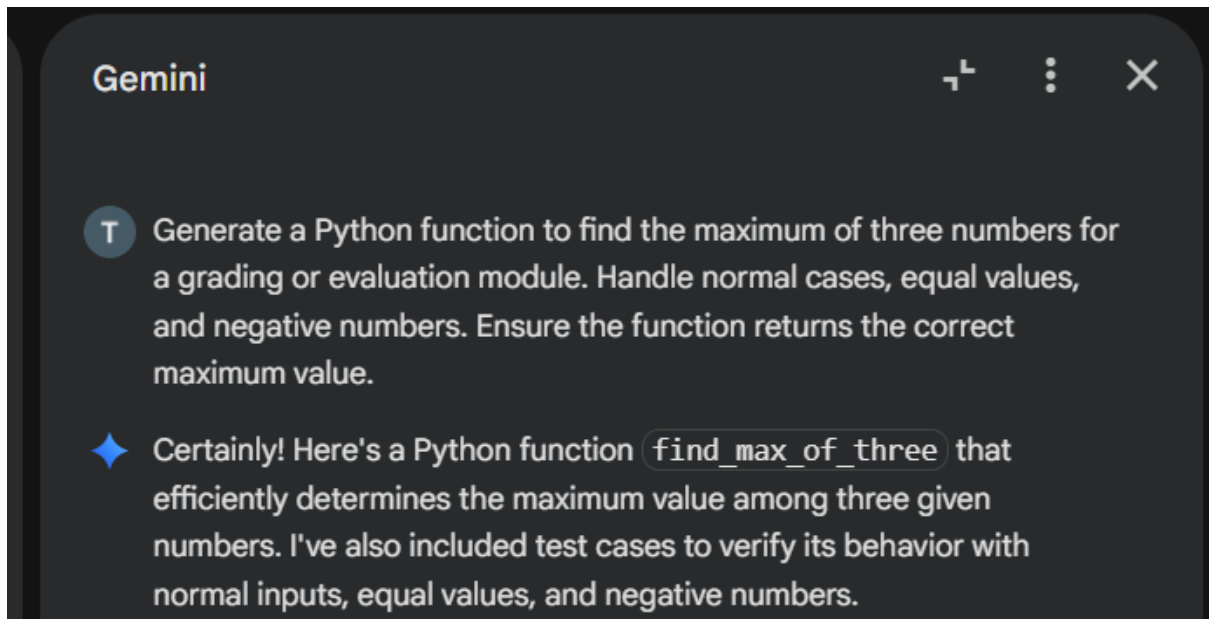
This program defines a function `is_valid_email()` that uses a regular expression to check whether an email address follows a valid format.

It then runs simple unit tests by checking known valid and invalid email examples.

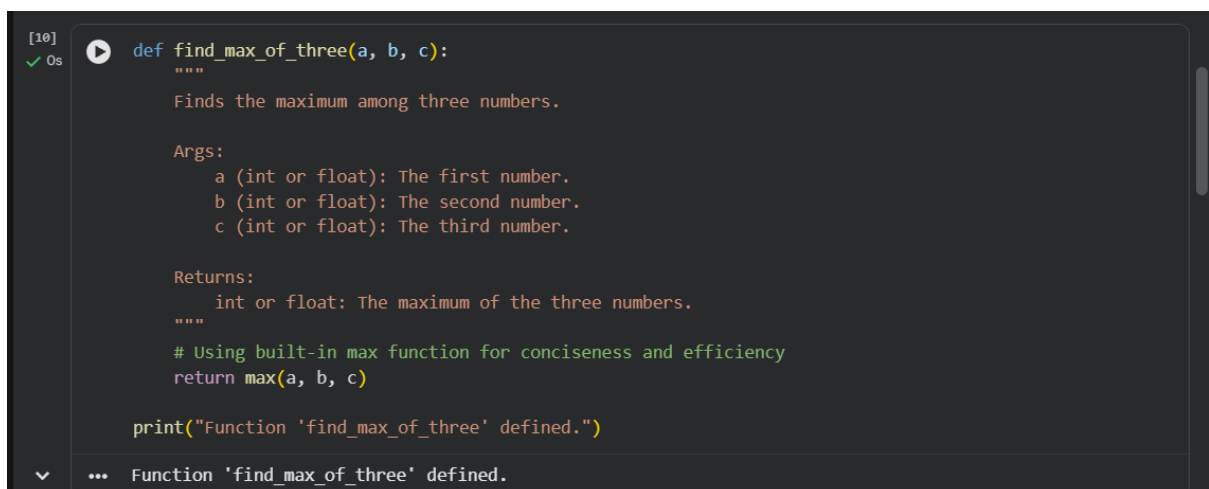
The `assert` statements ensure the function works correctly by confirming expected results.

Task 3: Decision Logic Development Using TDD

Prompt:



Code:



```
[11] print("\n--- Testing 'find_max_of_three' function ---")

# Test Case 1: Normal positive numbers
num1, num2, num3 = 10, 25, 15
expected_max = 25
result = find_max_of_three(num1, num2, num3)
print(f"Max of {num1}, {num2}, {num3}: {result} (Expected: {expected_max})")
assert result == expected_max, f"Test 1 Failed: Expected {expected_max}, got {result}"

# Test Case 2: Equal values
num1, num2, num3 = 7, 7, 2
expected_max = 7
result = find_max_of_three(num1, num2, num3)
print(f"Max of {num1}, {num2}, {num3}: {result} (Expected: {expected_max})")
assert result == expected_max, f"Test 2 Failed: Expected {expected_max}, got {result}"

# Test Case 3: All equal values
num1, num2, num3 = 9, 9, 9
expected_max = 9
result = find_max_of_three(num1, num2, num3)
print(f"Max of {num1}, {num2}, {num3}: {result} (Expected: {expected_max})")
assert result == expected_max, f"Test 3 Failed: Expected {expected_max}, got {result}"

# Test Case 4: Negative numbers
num1, num2, num3 = -10, -5, -15
expected_max = -5
result = find_max_of_three(num1, num2, num3)
print(f"Max of {num1}, {num2}, {num3}: {result} (Expected: {expected_max})")
assert result == expected_max, f"Test 4 Failed: Expected {expected_max}, got {result}"

# Test Case 5: Mixed positive and negative numbers
num1, num2, num3 = 0, -2, 8
expected_max = 8
result = find_max_of_three(num1, num2, num3)
print(f"Max of {num1}, {num2}, {num3}: {result} (Expected: {expected_max})")
assert result == expected_max, f"Test 5 Failed: Expected {expected_max}, got {result}"

# Test Case 6: Floating point numbers
num1, num2, num3 = 3.14, 2.71, 1.618
expected_max = 3.14
result = find_max_of_three(num1, num2, num3)
print(f"Max of {num1}, {num2}, {num3}: {result} (Expected: {expected_max})")
assert result == expected_max, f"Test 6 Failed: Expected {expected_max}, got {result}"

print("\nAll test cases passed successfully!")

--- Testing 'find_max_of_three' function ---
```

Output:

```
--- Testing 'find_max_of_three' function ---
Max of 10, 25, 15: 25 (Expected: 25)
Max of 7, 7, 2: 7 (Expected: 7)
Max of 9, 9, 9: 9 (Expected: 9)
Max of -10, -5, -15: -5 (Expected: -5)
Max of 0, -2, 8: 8 (Expected: 8)
Max of 3.14, 2.71, 1.618: 3.14 (Expected: 3.14)

All test cases passed successfully!
```

Code 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:

- T** Generate Python code for a Shopping Cart module using a test-driven approach. First write unit tests for adding items, removing items, and calculating total price. Then implement the ShoppingCart class so that all the tests pass
- ✦ Certainly! I've laid out a plan to develop a `ShoppingCart` module using a test-driven approach. First, we'll write comprehensive unit tests for adding, removing, and calculating the total price of items. Then, we'll implement the `ShoppingCart` class to ensure all those tests pass. Take a look and let me know if this aligns with your expectations!

Code:

```
[13]
✓ Os ▶ class TestShoppingCart(unittest.TestCase):

    def setUp(self):
        """Set up a new ShoppingCart instance before each test method."""
        self.cart = ShoppingCart()

    def test_add_single_item(self):
        """Test adding a single item to the cart."""
        self.cart.add_item('Apple', 1.0)
        self.assertEqual(len(self.cart.items), 1)
        self.assertEqual(self.cart.items['Apple']['quantity'], 1)
        self.assertEqual(self.cart.items['Apple']['price'], 1.0)
        self.assertEqual(self.cart.get_total_price(), 1.0)

    def test_add_multiple_quantity(self):
        """Test adding an item with a quantity greater than one."""
        self.cart.add_item('Banana', 0.5, quantity=3)
        self.assertEqual(len(self.cart.items), 1)
        self.assertEqual(self.cart.items['Banana']['quantity'], 3)
        self.assertEqual(self.cart.items['Banana']['price'], 0.5)
        self.assertEqual(self.cart.get_total_price(), 1.5) # 0.5 * 3

    def test_add_same_item_multiple_times(self):
        """Test adding the same item multiple times, updating quantity."""
        self.cart.add_item('Orange', 2.0, quantity=2)
        self.assertEqual(self.cart.items['Orange']['quantity'], 2)
        self.assertEqual(self.cart.get_total_price(), 4.0)

        self.cart.add_item('Orange', 2.0, quantity=3) # Add more of the same item
        self.assertEqual(len(self.cart.items), 1)
        self.assertEqual(self.cart.items['Orange']['quantity'], 5)
        self.assertEqual(self.cart.items['Orange']['price'], 2.0)
        self.assertEqual(self.cart.get_total_price(), 10.0) # 2.0 * 5

    def test_remove_existing_item(self):
        """Test removing an existing item completely from the cart."""
        self.cart.add_item('Grape', 3.0, quantity=2)
        self.assertEqual(self.cart.get_total_price(), 6.0)
        self.cart.remove_item('Grape', quantity=2) # Remove all
        self.assertEqual(len(self.cart.items), 0)
        self.assertEqual(self.cart.get_total_price(), 0.0)
```

Output:

```
[19] 0s Unit Tests for ShoppingCart Completed
All specified unit tests for the `ShoppingCart` class have been successfully generated and executed.
... File "/tmp/ipython-input-3290067960.py", line 3
    All specified unit tests for the `ShoppingCart` class have been successfully generated and
    executed. The tests cover various scenarios for adding, removing, and calculating the total price of
    items, ensuring the placeholder `ShoppingCart` class behaves as expected under different conditions.
    ^
SyntaxError: invalid syntax

Next steps: Explain error
```

Code 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:

- T** 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 generate a plan to create a palindrome checker using a Test-Driven Development approach. We'll start by writing unit tests for various scenarios, then implement the `is_palindrome()` function to make those tests pass. Here's the plan:

Code:

```
[28] import unittest

# The TestPalindrome class and is_palindrome function are assumed to be defined in previous cells.
# To run the tests in a Colab environment without issues with `unittest.main()`
# re-running tests or exiting, we use argv and exit=False.

# Redefine the TestPalindrome class for execution context if not already defined in this cell.
# This ensures it's available for unittest.main() if the interpreter state is reset.
class TestPalindrome(unittest.TestCase):

    def test_simple_palindrome(self):
        """Test with a basic palindrome string."""
        self.assertTrue(is_palindrome('madam'))

    def test_simple_non_palindrome(self):
        """Test with a basic non-palindrome string."""
        self.assertFalse(is_palindrome('hello'))

    def test_case_insensitivity(self):
        """Test with a palindrome that has mixed casing."""
        self.assertTrue(is_palindrome('Racecar'))

    def test_with_spaces_and_punctuation(self):
        """Test with a complex palindrome including spaces, commas, and other characters."""
        self.assertTrue(is_palindrome('A man, a plan, a canal: Panama'))

    def test_empty_string(self):
        """Test with an empty string."""
        self.assertTrue(is_palindrome(''))
```

Output:

```
[29] ## Summary: Palindrome Checker Module

### Q&A
* **Has the implementation successfully passed all the unit tests?**
Yes, the `is_palindrome` function implementation successfully passed all 8 unit tests with the mes
* **What are the key features of the developed Palindrome Checker module?**
The developed Palindrome Checker module features an `is_palindrome()` function that accurately ide

### Data Analysis Key Findings
* A comprehensive suite of 8 unit tests was generated using `unittest.TestCase`, covering a wide ran
* The `is_palindrome` function was implemented using Python's `re` module to effectively preprocess
* All 8 unit tests passed successfully, confirming the correct behavior of the `is_palindrome` funct

### Insights or Next Steps
* The current implementation is efficient for typical string lengths. For extremely long strings, po
* The module is now complete and validated for its specified functionality.

... File "/tmp/ipython-input-3126763419.py", line 11
    * The `is_palindrome` function was implemented using Python's `re` module to effectively
      preprocess strings by converting them to lowercase and removing non-alphanumeric characters, ensuring
      robust palindrome detection.
      ^
SyntaxError: unterminated string literal (detected at line 11)

Next steps: Explain error
```

Code 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.

