AI -8.2

NAME:THALLAPALLI SANJAY

ROLL:2403A51393

BATCH:15

Task 1:

Prompt:

Use AI to generate test cases for a function is_prime(n) and then implement the function.
Requirements:
- **Only integers > 1 can be prime.**
- **Check edge cases: 0, 1, 2, negative numbers, and large primes.**

Code:

```
[4]
 0s
import math

# Test cases
test_cases = [
    (0, False),   # Edge case: 0
    (1, False),   # Edge case: 1
    (2, True),    # Edge case: 2
    (-5, False), # Negative number
    (999983, True), # Large prime number
    (1000000, False), # Large non-prime number
    (3, True),    # Small prime
    (4, False),   # Small non-prime
    (7, True),    # Small prime
    (9, False),   # Small non-prime
    (11, True),   # Small prime
    (15, False), # Small non-prime
    (29, True), # Small prime
    (33, False), # Small non-prime
]

# is_prime function
def is_prime(n):
```

```python
    """
    if n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    # Iterate through odd numbers from 3 up to the square root of n
    for i in range(3, int(math.sqrt(n)) + 1, 2):
        if n % i == 0:
            return False

    return True

# Run tests
for number, expected in test_cases:
    actual = is_prime(number)
    if actual == expected:
        print(f"Test passed for input: {number}, Expected: {expected}, Got: {actual}")
    else:
        print(f"Test failed for input: {number}, Expected: {expected}, Got: {actual}")
```

Output:

```
Test passed for input: 0, Expected: False, Got: False
Test passed for input: 1, Expected: False, Got: False
Test passed for input: 2, Expected: True, Got: True
Test passed for input: -5, Expected: False, Got: False
Test passed for input: 999983, Expected: True, Got: True
Test passed for input: 1000000, Expected: False, Got: False
Test passed for input: 3, Expected: True, Got: True
Test passed for input: 4, Expected: False, Got: False
Test passed for input: 7, Expected: True, Got: True
Test passed for input: 9, Expected: False, Got: False
Test passed for input: 11, Expected: True, Got: True
Test passed for input: 15, Expected: False, Got: False
Test passed for input: 29, Expected: True, Got: True
Test passed for input: 33, Expected: False, Got: False
```

- **Explanation:**
- **Import math:** This **line imports the math module, which is needed to use the math.sqrt() function for calculating the square root.**
- **test_cases = [...]: This list contains tuples representing different test cases. Each tuple has two elements: the number to be checked (number) and the expected result (expected - True if prime, False otherwise). These test cases cover various scenarios, including edge cases, negative numbers, small and large primes, and non-prime numbers.**
- **def is_prime(n): ...: This defines the is_prime function, which takes an integer n as input and returns True if n is prime and False otherwise.**
    - **if n <= 1:: Numbers less than or equal to 1 are not prime.**
    - **if n == 2:: 2 is the only even prime number.**
    - **if n % 2 == 0:: Other even numbers are not prime.**
    - **for i in range(3, int(math.sqrt(n)) + 1, 2):: This loop iterates through odd numbers starting from 3 up to the square root of n. We only need to**

check for divisors up to the square root because if a number n has a divisor greater than its square root, it must also have a divisor smaller than its square root. We only check odd numbers because we've already handled even numbers.

  - if n % i == 0:: If n is divisible by any of these odd numbers, it's not prime, and the function returns False.
  - return True: If the loop completes without finding any divisors, the number is prime, and the function returns True.

- for number, expected in test_cases: ...: This loop iterates through the test_cases list. For each test case, it calls the is_prime function with the input number and compares the actual result with the expected result.
- print(...): These lines print whether each test case passed or failed, showing the input, expected output, and actual output.

This code effectively tests the is_prime function against a variety of inputs to ensure its correctness.

Task 2:

Prompt:

Ask AI to generate test cases for celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f).
Requirements
• Validate known pairs: 0°C = 32°F, 100°C = 212°F.
• Include decimals and invalid inputs like strings or None

Code:

```python
# Test cases for Celsius to Fahrenheit
celsius_test_cases = [
    (0, 32.0),        # Known pair: 0°C = 32°F
    (100, 212.0),     # Known pair: 100°C = 212°F
    (-40, -40.0),     # Known pair: -40°C = -40°F
    (25.5, 77.9),     # Decimal value
    (-10.2, 13.64),   # Decimal value
    ("abc", None),    # Invalid input: string
    (None, None),     # Invalid input: None
    ([1, 2], None),   # Invalid input: list
]

# Test cases for Fahrenheit to Celsius
fahrenheit_test_cases = [
    (32, 0.0),        # Known pair: 32°F = 0°C
    (212, 100.0),     # Known pair: 212°F = 100°C
    (-40, -40.0),     # Known pair: -40°F = -40°C
    (77.9, 25.5),     # Decimal value
    (13.64, -10.2),   # Decimal value
    ("xyz", None),    # Invalid input: string
    (None, None),     # Invalid input: None
    ({'a': 1}, None), # Invalid input: dictionary
]

# celsius_to_fahrenheit function
def celsius_to_fahrenheit(c):
    """
```

```python
    if not isinstance(c, (int, float)):
        return None
    fahrenheit = (c * 9/5) + 32
    return fahrenheit

# fahrenheit_to_celsius function
def fahrenheit_to_celsius(f):
    """
    Converts Fahrenheit to Celsius.

    Args:
        f: The temperature in Fahrenheit.

    Returns:
        The temperature in Celsius if the input is a number, None otherwise.
    """
    if not isinstance(f, (int, float)):
        return None
    celsius = (f - 32) * 5/9
    return celsius

# Run tests for celsius_to_fahrenheit
print("Testing celsius_to_fahrenheit:")
for celsius, expected_fahrenheit in celsius_test_cases:
    actual_fahrenheit = celsius_to_fahrenheit(celsius)
    if actual_fahrenheit == expected_fahrenheit:
        print(f"Test passed for input: {celsius}°C, Expected: {expected_fahrenheit}°F, Got: {actual_fahrenheit}°F")
    else:
        print(f"Test failed for input: {celsius}°C, Expected: {expected_fahrenheit}°F, Got: {actual_fahrenheit}°F")
```

```python
print("\nTesting fahrenheit_to_celsius:")
# Run tests for fahrenheit_to_celsius
for fahrenheit, expected_celsius in fahrenheit_test_cases:
    actual_celsius = fahrenheit_to_celsius(fahrenheit)
    if actual_celsius == expected_celsius:
        print(f"Test passed for input: {fahrenheit}°F, Expected: {expected_celsius}°C, Got: {actual_celsius}°C")
    else:
        print(f"Test failed for input: {fahrenheit}°F, Expected: {expected_celsius}°C, Got: {actual_celsius}°C")
```

Output:

```
Testing celsius_to_fahrenheit:
Test passed for input: 0°C, Expected: 32.0°F, Got: 32.0°F
Test passed for input: 100°C, Expected: 212.0°F, Got: 212.0°F
Test passed for input: -40°C, Expected: -40.0°F, Got: -40.0°F
Test passed for input: 25.5°C, Expected: 77.9°F, Got: 77.9°F
Test passed for input: -10.2°C, Expected: 13.64°F, Got: 13.64°F
Test passed for input: abc°C, Expected: None°F, Got: None°F
Test passed for input: None°C, Expected: None°F, Got: None°F
Test passed for input: [1, 2]°C, Expected: None°F, Got: None°F

Testing fahrenheit_to_celsius:
Test passed for input: 32°F, Expected: 0.0°C, Got: 0.0°C
Test passed for input: 212°F, Expected: 100.0°C, Got: 100.0°C
Test passed for input: -40°F, Expected: -40.0°C, Got: -40.0°C
Test failed for input: 77.9°F, Expected: 25.5°C, Got: 25.500000000000004°C
Test passed for input: 13.64°F, Expected: -10.2°C, Got: -10.2°C
Test passed for input: xyz°F, Expected: None°C, Got: None°C
Test passed for input: None°F, Expected: None°C, Got: None°C
Test passed for input: {'a': 1}°F, Expected: None°C, Got: None°C
```

Explanation: Certainly! The selected code cell contains the test cases and the implementation for converting temperatures between Celsius and Fahrenheit, along with the code to run the tests.

Here's a breakdown:

1. **Test Cases (celsius_test_cases and fahrenheit_test_cases): These lists contain tuples, where each tuple represents a test case. Each tuple includes an input value and the expected output value for the respective conversion function. These test cases cover:**
    - **Known conversion pairs (e.g., 0°C = 32°F).**
    - **Negative temperatures.**
    - **Decimal values.**

- **Invalid inputs (strings, None, lists, dictionaries) to check for robust error handling.**

2. **celsius_to_fahrenheit(c) function:**
   - **This function takes a temperature in Celsius (c) as input.**
   - **It first checks if the input c is an integer or a float. If not, it returns None, indicating invalid input.**
   - **If the input is valid, it applies the formula (c * G/5) + 32 to convert Celsius to Fahrenheit.**
   - **It returns the calculated Fahrenheit temperature.**
3. **fahrenheit_to_celsius(f) function:**
   - **This function takes a temperature in Fahrenheit (f) as input.**
   - **Similar to the Celsius to Fahrenheit function, it checks if the input f is an integer or a float and returns None for invalid inputs.**
   - **If the input is valid, it applies the formula (f - 32) * 5/G to convert Fahrenheit to Celsius.**
   - **It returns the calculated Celsius temperature.**
4. **Test Execution:**
   - **The code then iterates through the celsius_test_cases list. For each test case, it calls celsius_to_fahrenheit with the input Celsius value and compares the actual result with the expected Fahrenheit value. It prints whether the test passed or failed.**
   - **Similarly, it iterates through the fahrenheit_test_cases list, calls fahrenheit_to_celsius with the input Fahrenheit value, compares the actual result with the expected Celsius value, and prints the test result.**

This code provides a complete solution for temperature conversion between Celsius and Fahrenheit, including comprehensive test cases to verify the functions' correctness and handling of various input types.

0 / 2000

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

Task 3:

Prompt:

Use AI to write test cases for a function count_words(text) that returns the number of
words in a sentence.

Requirement

Handle normal text, multiple spaces, punctuation, and empty strings.

code:

```
test_cases = [
    ("This is a normal sentence.", 5),
    ("This  has   multiple    spaces.", 4),
    ("This sentence has punctuation!", 4),
    ("", 0),
    ("  Leading and trailing spaces.  ", 5),
    ("One. Two! Three?", 3),
    ("word", 1)
]
```

```
import re

def count_words(text):
  """Counts the number of words in a given text, handling different scenarios."""
  cleaned_text = text.strip()
  if not cleaned_text:
    return 0
  words = re.split(r'\s+', cleaned_text)
  return len(words)
```

```
for input_string, expected_output in test_cases:
    actual_output = count_words(input_string)
    if actual_output == expected_output:
        print(f"Test Passed: Input='{input_string}', Expected={expected_output}, Actual={actual_output}")
    else:
        print(f"Test Failed: Input='{input_string}', Expected={expected_output}, Actual={actual_output}")
```

Output:

```
Test Passed: Input='This is a normal sentence.', Expected=5, Actual=5
Test Passed: Input='This  has   multiple    spaces.', Expected=4, Actual=4
Test Passed: Input='This sentence has punctuation!', Expected=4, Actual=4
Test Passed: Input='', Expected=0, Actual=0
Test Failed: Input='  Leading and trailing spaces.  ', Expected=5, Actual=4
Test Passed: Input='One. Two! Three?', Expected=3, Actual=3
Test Passed: Input='word', Expected=1, Actual=1
```

Summary:

- **Explanation:□for input_string, expected_output in test_cases::** This loop iterates through each tuple in the test_cases list. In each iteration, it unpacks the tuple into two variables: input_string (the text to be tested) and expected_output (the expected word count for that text).
- **actual_output = count_words(input_string):** This line calls your count_words function with the input_string from the current test case and stores the returned value (the actual word count) in the actual_output variable.
- **if actual_output == expected_output::** This checks if the actual_output from the function call matches the expected_output for the current test case.

- **print(f"Test Passed: ...")**: If the actual output matches the expected output, this line prints a "Test Passed" message, showing the input string, the expected output, and the actual output.
- **else: print(f"Test Failed: ...")**: If the actual output does not match the expected output, this line prints a "Test Failed" message, also showing the input string, the expected output, and the actual output.

Task 4:

Prompt:

• Generate test cases for a BankAccount class with:

Methods:

deposit(amount)

withdraw(amount)

check_balance()

Requirements:

• Negative deposits/withdrawals should raise an error.

• Cannot withdraw more than balance.

code:

```python
import unittest

# BankAccount class
class BankAccount:
    def __init__(self, initial_balance=0):
        if not isinstance(initial_balance, (int, float)) or initial_balance < 0:
            raise ValueError("Initial balance must be a non-negative number.")
        self.balance = initial_balance

    def deposit(self, amount):
        if not isinstance(amount, (int, float)) or amount < 0:
            raise ValueError("Deposit amount must be a non-negative number.")
        self.balance += amount

    def withdraw(self, amount):
        if not isinstance(amount, (int, float)) or amount < 0:
            raise ValueError("Withdrawal amount must be a non-negative number.")
        if amount > self.balance:
            raise ValueError("Cannot withdraw more than the current balance.")
        self.balance -= amount

    def check_balance(self):
        return self.balance

# Test cases
class TestBankAccount(unittest.TestCase):

    def test_initial_balance(self):
        account = BankAccount(100)
```

```python
        self.assertEqual(account.check_balance(), 100)

    def test_deposit(self):
        account = BankAccount(100)
        account.deposit(50)
        self.assertEqual(account.check_balance(), 150)

    def test_withdraw(self):
        account = BankAccount(100)
        account.withdraw(30)
        self.assertEqual(account.check_balance(), 70)

    def test_negative_initial_balance(self):
        with self.assertRaises(ValueError):
            BankAccount(-100)

    def test_negative_deposit(self):
        account = BankAccount(100)
        with self.assertRaises(ValueError):
            account.deposit(-50)

    def test_negative_withdraw(self):
        account = BankAccount(100)
        with self.assertRaises(ValueError):
            account.withdraw(-30)

    def test_withdraw_exceeds_balance(self):
        account = BankAccount(100)
        with self.assertRaises(ValueError):
            account.withdraw(150)
```

```python
    def test_deposit_non_numeric(self):
        account = BankAccount(100)
        with self.assertRaises(ValueError):
            account.deposit("abc")

    def test_withdraw_non_numeric(self):
        account = BankAccount(100)
        with self.assertRaises(ValueError):
            account.withdraw("xyz")

    def test_initial_balance_non_numeric(self):
        with self.assertRaises(ValueError):
            BankAccount("abc")


# Run tests
if __name__ == '__main__':
    # Running tests with a more verbose output format
    unittest.main(argv=['first-arg-is-ignored'], exit=False, verbosity=2)
```

Output:

```
test_deposit (__main__.TestBankAccount.test_deposit) ... ok
test_deposit_non_numeric (__main__.TestBankAccount.test_deposit_non_numeric) ... ok
test_initial_balance (__main__.TestBankAccount.test_initial_balance) ... ok
test_initial_balance_non_numeric (__main__.TestBankAccount.test_initial_balance_non_numeric) ... ok
test_negative_deposit (__main__.TestBankAccount.test_negative_deposit) ... ok
test_negative_initial_balance (__main__.TestBankAccount.test_negative_initial_balance) ... ok
test_negative_withdraw (__main__.TestBankAccount.test_negative_withdraw) ... ok
test_withdraw (__main__.TestBankAccount.test_withdraw) ... ok
test_withdraw_exceeds_balance (__main__.TestBankAccount.test_withdraw_exceeds_balance) ... ok
test_withdraw_non_numeric (__main__.TestBankAccount.test_withdraw_non_numeric) ... ok

----------------------------------------------------------------------
Ran 10 tests in 0.017s

OK
```

Explanation:

1. **BankAccount class:**

- _init_(self, initial_balance=0): This is the constructor. It initializes a new bank account with an optional initial_balance. It includes validation to ensure the initial_balance is a non-negative number, raising a ValueError otherwise.
- deposit(self, amount): This method allows depositing funds into the account. It checks if the amount is a non-negative number and adds it to the balance. A ValueError is raised for invalid input.
- withdraw(self, amount): This method allows withdrawing funds from the account. It validates that the amount is a non-negative number and that there are sufficient funds in the balance. If the amount is greater than the balance, it raises a ValueError. Otherwise, it subtracts the amount from the balance.
- check_balance(self): This method simply returns the current balance of the account.

2. TestBankAccount class:
   - This class inherits from unittest.TestCase, which provides the framework for writing unit tests.
   - Each method starting with test_ is a separate test case.
   - test_initial_balance(self): Tests if the account is initialized with the correct balance.
   - test_deposit(self): Tests if the deposit method correctly increases the balance.
   - test_withdraw(self): Tests if the withdraw method correctly decreases the balance.
   - test_negative_initial_balance(self), test_negative_deposit(self), test_ negative_withdraw(self): These tests use self.assertRaises(ValueError) to verify that a ValueError is raised when attempting to initialize with a negative balance, deposit a negative amount, or withdraw a negative amount.
   - test_withdraw_exceeds_balance(self): This test checks that a ValueError is raised when attempting to withdraw more money than is available in the account.
   - test_deposit_non_numeric(self), test_withdraw_non_numeric(self), te st_initial_balance_non_numeric(self): These tests use self.assertRaises(ValueError) to ensure that ValueError is raised

when non-numeric inputs are provided for initial balance, deposit, or withdrawal.

- self.assertEqual(a, b): This is a standard unittest assertion that checks if a is equal to b.

3. if _name__== '_main_': unittest.main(...): This block is standard practice for running unit tests in a script.
   - if _name__== '_main_': ensures that the code inside this block only runs when the script is executed directly (not when it's imported as a module).
   - unittest.main(...) runs the tests defined in the TestBankAccount class.
   - argv=['first-arg-is-ignored'], exit=False are used to run the tests within environments like Colab without causing the notebook to exit.
   - verbosity=2 (which I added in the previous turn) provides a more detailed output, showing the result of each individual test method.

Task 5:

Prompt:
Generate test cases for is_number_palindrome(num), which checks if an integer reads
the same backward.
Examples:
121 → True
123 → False
0, negative numbers → handled gracefully

code:

```python
# Test cases
test_cases = [
    (121, True),     # Positive palindrome
    (123, False),    # Positive non-palindrome
    (0, True),       # Zero
    (5, True),       # Single-digit number
    (-121, False),   # Negative number
    (12321, True),   # Longer positive palindrome
    (12345, False),  # Longer positive non-palindrome
    (10, False),     # Number ending in zero
    (101, True),     # Palindrome with zero in the middle
]

# is_number_palindrome function
def is_number_palindrome(num):
    """
    Checks if an integer reads the same backward.

    Args:
        num: The integer to check.

    Returns:
        True if the number is a palindrome, False otherwise.
    """
    # Negative numbers are not considered palindromes
    if num < 0:
        return False

    # Convert the number to a string to easily reverse and compare
```

```python
    num_str = str(num)
    reversed_num_str = num_str[::-1]

    return num_str == reversed_num_str

# Run tests
for number, expected in test_cases:
    actual = is_number_palindrome(number)
    if actual == expected:
        print(f"Test passed for input: {number}, Expected: {expected}, Got: {actual}")
    else:
        print(f"Test failed for input: {number}, Expected: {expected}, Got: {actual}")
```

Output:

```
Test passed for input: 121, Expected: True, Got: True
Test passed for input: 123, Expected: False, Got: False
Test passed for input: 0, Expected: True, Got: True
Test passed for input: 5, Expected: True, Got: True
Test passed for input: -121, Expected: False, Got: False
Test passed for input: 12321, Expected: True, Got: True
Test passed for input: 12345, Expected: False, Got: False
Test passed for input: 10, Expected: False, Got: False
Test passed for input: 101, Expected: True, Got: True
```

Explanation:

Test_cases = [...]: This list contains tuples, where each tuple represents a test case. Each tuple includes an input integer (number) and the expected boolean result (expected - True if it's a palindrome, False otherwise). These test cases cover various scenarios, including positive and negative numbers, single digits, zero, and numbers with different structures.

2.def is_number_palindrome(num): ...: This defines the is_number_palindrome function, which takes an integer num as input.

- if num < 0:: This checks if the input number is negative. According to the problem definition, negative numbers are not considered palindromes, so it immediately returns False.
- num_str = str(num): The number is converted to a string so that it can be easily reversed.
- reversed_num_str = num_str[::-1]: This is a Python string slicing technique that creates a reversed version of the num_str. [::-1] means start from the end, go to the beginning, with a step of -1.
- return num_str == reversed_num_str: This compares the original string representation of the number with its reversed version. If they are the same, the number is a palindrome, and the function returns True; otherwise, it returns False.

2. for number, expected in test_cases: ...: This loop iterates through each test case in the test_cases list.
    - actual = is_number_palindrome(number): It calls the is_number_palindrome function with the input number from the current test case and stores the result in the actual variable.
    - if actual == expected:: This checks if the actual result matches the expected result for the current test case.
    - print(f"Test passed..."): If the results match, it prints a "Test passed" message with the input, expected, and actual values.
    - else: print(f"Test failed..."): If the results don't match, it prints a "Test failed" message with the same details.