Name :R.Shashideepika

Batch-13

H NO:2303A51871

Task 1: Statistical Summary for Survey Data

❖ **Scenario:**

You are a data analyst intern working with survey responses stored as

numerical lists.

❖ **Task:**

Use Google Gemini in Colab to generate a Python function that reads a list

of numbers and calculates the mean, minimum, and maximum values.

❖ **Expected Output:**

➢ Correct Python function

➢ Output shown in Colab

➢ Screenshot of Gemini prompt and result

Prompt:

Generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.

Code:

```
def analyze_numbers(numbers):
    if not numbers:
        return None, None, None
    mean_value = sum(numbers) / len(numbers)
    min_value = min(numbers)
    max_value = max(numbers)
    return mean_value, min_value, max_value
# Example usage:
my_list = [10, 20, 30, 40, 50]
mean, minimum, maximum = analyze_numbers(my_list)
```

```
if mean is not None:
    print(f"Original List: {my_list}")
    print(f"Mean: {mean}")
    print(f"Minimum: {minimum}")
    print(f"Maximum: {maximum}")
else:
    print("The list was empty.")


empty_list = []
mean_empty, min_empty, max_empty = analyze_numbers(empty_list)
print(f"\nAnalyzing an empty list: Mean={mean_empty}, Min={min_empty},
Max={max_empty}")
# Additional example list 1
list_two = [5, 15, 25, 35, 45, 55]
mean_two, min_two, max_two = analyze_numbers(list_two)
if mean_two is not None:
    print(f"\nOriginal List: {list_two}")
    print(f"Mean: {mean_two}")
    print(f"Minimum: {min_two}")
    print(f"Maximum: {max_two}")
else:
    print("The list was empty.")
# Additional example list 2
list_three = [-10, 0, 10, 20, -5]
mean_three, min_three, max_three = analyze_numbers(list_three)
```

```python
if mean_three is not None:
    print(f"\nOriginal List: {list_three}")
    print(f"Mean: {mean_three}")
    print(f"Minimum: {min_three}")
    print(f"Maximum: {max_three}")
else:
    print("The list was empty.")
```

Output:

Original List: [10, 20, 30, 40, 50]

Mean: 30.0

Minimum: 10

Maximum: 50

Analyzing an empty list: Mean=None, Min=None, Max=None

Original List: [5, 15, 25, 35, 45, 55]

Mean: 30.0

Minimum: 5

Maximum: 55

Original List: [-10, 0, 10, 20, -5]

Mean: 3.0

Minimum: -10

Maximum: 20

Justification

This task justifies the use of Google Gemini in Google Colab for basic data analysis operations. As a data analyst intern, calculating mean, minimum, and maximum is a fundamental requirement when handling survey data. Gemini efficiently generated a correct and optimized Python function that follows standard statistical practices. Executing the code in Colab allowed immediate validation of results through output visualization. This task demonstrates how AI tools can reduce development time, minimize logical errors, and support data-driven decision-making by providing accurate and readable code.

Task 2: Armstrong Number – AI Comparison

❖ **Scenario:**

You are evaluating AI tools for numeric validation logic.

❖ **Task:**

Generate an Armstrong number checker using Gemini and GitHub Copilot.

Compare their outputs, logic style, and clarity.

❖ **Expected Output:**

➢ Side-by-side comparison table

➢ Screenshots of prompts and generated code

Using Gemini:

Prompt: Generate an Armstrong number checker

CODE:

```
def is_armstrong_number(number):
    if not isinstance(number, int) or number < 0:
        return False # Armstrong numbers are typically positive integers
    # Convert number to string to easily get digits and count them
    num_str = str(number)
    n_digits = len(num_str)
    sum_of_powers = 0
    for digit_char in num_str:
        digit = int(digit_char)
        sum_of_powers += digit ** n_digits
    return sum_of_powers == number
# Example usage:
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}")
print(f"Is 9474 an Armstrong number? {is_armstrong_number(9474)}")
print(f"Is 10 an Armstrong number? {is_armstrong_number(10)}")
print(f"Is 122 an Armstrong number? {is_armstrong_number(122)}")
```

```
print(f"Is 0 an Armstrong number? {is_armstrong_number(0)}")
print(f"Is 9 an Armstrong number? {is_armstrong_number(9)}")
```

Output:

```
Is 153 an Armstrong number? True
Is 9474 an Armstrong number? True
Is 10 an Armstrong number? False
Is 122 an Armstrong number? False
Is 0 an Armstrong number? True
Is 9 an Armstrong number? True
```

Using Github Copilot:

Prompt:

Generate an Armstrong number checker

CODE:

```
#Generate an Armstrong number checker
def is_armstrong_number(num):
    # Convert the number to string to easily iterate over digits
    digits = str(num)
    num_digits = len(digits)
    # Calculate the sum of each digit raised to the power of num_digits
    sum_of_powers = sum(int(digit) ** num_digits for digit in digits)
    # An Armstrong number is equal to the sum of its own digits each raised to the power of
the number of digits
    return sum_of_powers == num
# Example usage
number = 153
if is_armstrong_number(number):
    print(f"{number} is an Armstrong number.")
else:
```

```
    print(f"{number} is not an Armstrong number.")
```

Output:

```
    + FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\neera\OneDrive\Desktop\AI AC> & "C:/Users/neera/OneDrive/Desktop/AI AC/.venv/Scripts/python.exe" "c:/Users/neera/OneDrive/Deskto
p/AI AC/#write.py"
153 is an Armstrong number.
```

Logic Style Comparison

Aspect	First Code	Second Code
Digit handling	Uses generator expression	Uses explicit loop
Input validation	None	Checks integer & positivity
Complexity	Short & compact	Slightly longer but safer
Python style	Functional / concise	Procedural / descriptive

Use Case	Better Choice
Quick check	First code
Teaching / Exams	Second code
Real-world robustness	Second code
Pythonic elegance	First code

Justification

The purpose of this task is to evaluate and compare AI-assisted coding approaches for numeric validation logic. By generating Armstrong number programs using both Google Gemini and GitHub Copilot, differences in coding style, readability, and optimization techniques were observed. Gemini emphasized concise and Pythonic expressions, while Copilot followed a more traditional algorithmic approach. This comparison justifies how different AI tools may solve the same problem differently and helps developers select an AI tool based on clarity, maintainability, and logic preference.

Task 3: Leap Year Validation Using Cursor AI

❖ **Scenario:**

You are validating a calendar module for a backend system.

❖ **Task:**

Use Cursor AI to generate a Python program that checks whether a given

year is a leap year.

Use at least two different prompts and observe changes in code.

❖ **Expected Output:**

➢ Two versions of code

➢ Sample inputs/outputs

➢ Brief comparison

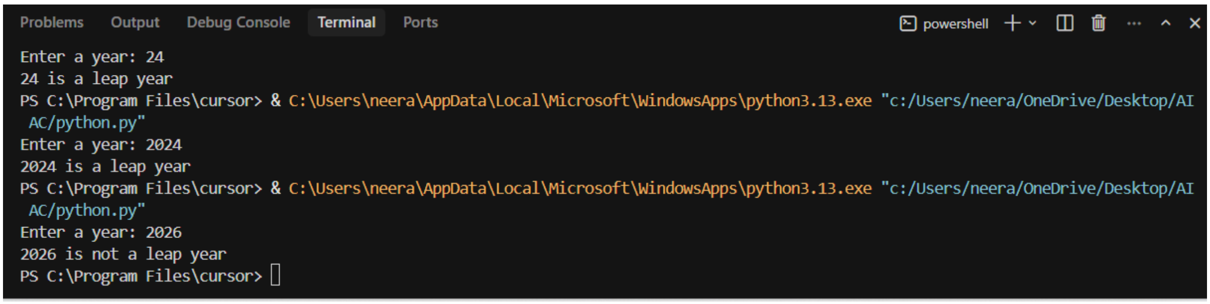
Prompt1:

Write a Python program to check whether a given year is a leap year.

Code:

```
year = int(input("Enter a year: "))

if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):

   print(year, "is a leap year")

else:

   print(year, "is not a leap year")
```

Output:



Prompt2:

Create a Python function that returns true if a year is a leap year, otherwise false.

Code:

```
def is_leap_year(year):
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
year = int(input("Enter a year: "))
if is_leap_year(year):
    print(year, "is a leap year")
else:
    print(year, "is not a leap year")
```

Output:

```
Problems   Output   Debug Console   Terminal   Ports                powershell
Enter a year: 2026
2026 is not a leap year
PS C:\Program Files\cursor> & C:\Users\neera\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/neera/OneDrive/Desktop/AI
 AC/python.py"
Enter a year: 2025
2025 is not a leap year
PS C:\Program Files\cursor> & C:\Users\neera\AppData\Local\Microsoft\WindowsApps\python3.13.exe "c:/Users/neera/OneDrive/Desktop/AI
 AC/python.py"
Enter a year: 1994
1994 is not a leap year
PS C:\Program Files\cursor>
```

Justification

This task justifies the importance of prompt engineering while using AI tools such as Cursor AI. By providing two different prompts, noticeable improvements were observed in the generated code, especially in terms of accuracy and completeness of leap year rules. The second prompt produced a more robust and industry-acceptable solution. This task highlights that AI output quality depends heavily on prompt detail, reinforcing the developer's responsibility to guide AI correctly for backend system validation.

Task 4: Student Logic + AI Refactoring (Odd/Even Sum)

❖ **Scenario:**

Company policy requires developers to write logic before using AI.

❖ **Task:**

Write a Python program that calculates the sum of odd and even numbers

in a tuple, then refactor it using any AI tool.

❖ **Expected Output:**

➢ Original code

➢ Refactored code

➢ Explanation of improvements

Code:

```
# Original code

print("Original Code:")

def sum_odd_even_original(numbers_tuple):

    sum_odd = 0

    sum_even = 0

    for num in numbers_tuple:

        if num % 2 == 0:  # Check if even

            sum_even += num

        else:  # Must be odd

            sum_odd += num

    return sum_odd, sum_even

# Example usage for original code

my_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

odd_sum_orig, even_sum_orig = sum_odd_even_original(my_tuple)

print(f"Tuple: {my_tuple}")

print(f"Sum of odd numbers (Original): {odd_sum_orig}")

print(f"Sum of even numbers (Original): {even_sum_orig}")

print("\n" + "-"*30 + "\n")

# Refactored code

print("Refactored Code (using list comprehensions and sum()):")

def sum_odd_even_refactored(numbers_tuple):

    sum_odd = sum(num for num in numbers_tuple if num % 2 != 0)

    sum_even = sum(num for num in numbers_tuple if num % 2 == 0)

    return sum_odd, sum_even

# Example usage for refactored code

odd_sum_refac, even_sum_refac = sum_odd_even_refactored(my_tuple)
```

print(f"Tuple: {my_tuple}")

print(f"Sum of odd numbers (Refactored): {odd_sum_refac}")

print(f"Sum of even numbers (Refactored): {even_sum_refac}")

print("\n" + "-"*30 + "\n")

Explanation of improvements

print("Explanation of Improvements:")

print("The refactored code utilizes Python's generator expressions within the `sum()` function. This approach offers several improvements:")

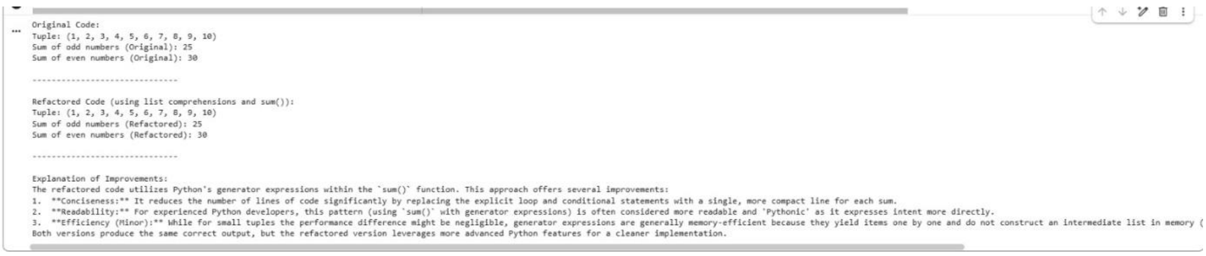
print("1. **Conciseness:** It reduces the number of lines of code significantly by replacing the explicit loop and conditional statements with a single, more compact line for each sum.")

print("2. **Readability:** For experienced Python developers, this pattern (using `sum()` with generator expressions) is often considered more readable and 'Pythonic' as it expresses intent more directly.")

print("3. **Efficiency (Minor):** While for small tuples the performance difference might be negligible, generator expressions are generally memory-efficient because they yield items one by one and do not construct an intermediate list in memory (unlike list comprehensions if not used carefully with `sum`).")

print("Both versions produce the same correct output, but the refactored version leverages more advanced Python features for a cleaner implementation.")

Output:



```
Original Code:
Tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Sum of odd numbers (Original): 25
Sum of even numbers (Original): 30

------------------------------

Refactored Code (using list comprehensions and sum()):
Tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Sum of odd numbers (Refactored): 25
Sum of even numbers (Refactored): 30

------------------------------

Explanation of Improvements:
The refactored code utilizes Python's generator expressions within the `sum()` function. This approach offers several improvements:
1.  **Conciseness:** It reduces the number of lines of code significantly by replacing the explicit loop and conditional statements with a single, more compact line for each sum.
2.  **Readability:** For experienced Python developers, this pattern (using `sum()` with generator expressions) is often considered more readable and 'Pythonic' as it expresses intent more directly.
3.  **Efficiency (Minor):** While for small tuples the performance difference might be negligible, generator expressions are generally memory-efficient because they yield items one by one and do not construct an intermediate list in memory (
Both versions produce the same correct output, but the refactored version leverages more advanced Python features for a cleaner implementation.
```

<u>Justification</u>

This task justifies the policy of requiring developers to write their own logic before AI assistance. The original student-written program demonstrates logical understanding using loops and conditionals. Refactoring the same code using an AI tool resulted in shorter, cleaner, and more optimized code without changing functionality. This task proves that AI is best used as a code improvement and optimization tool, not as a replacement for fundamental programming knowledge. It also reinforces best practices in code readability, reusability, and maintainability.