AI-Assisted Coding

Week-2.1

2303A51018

Batch-28

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.

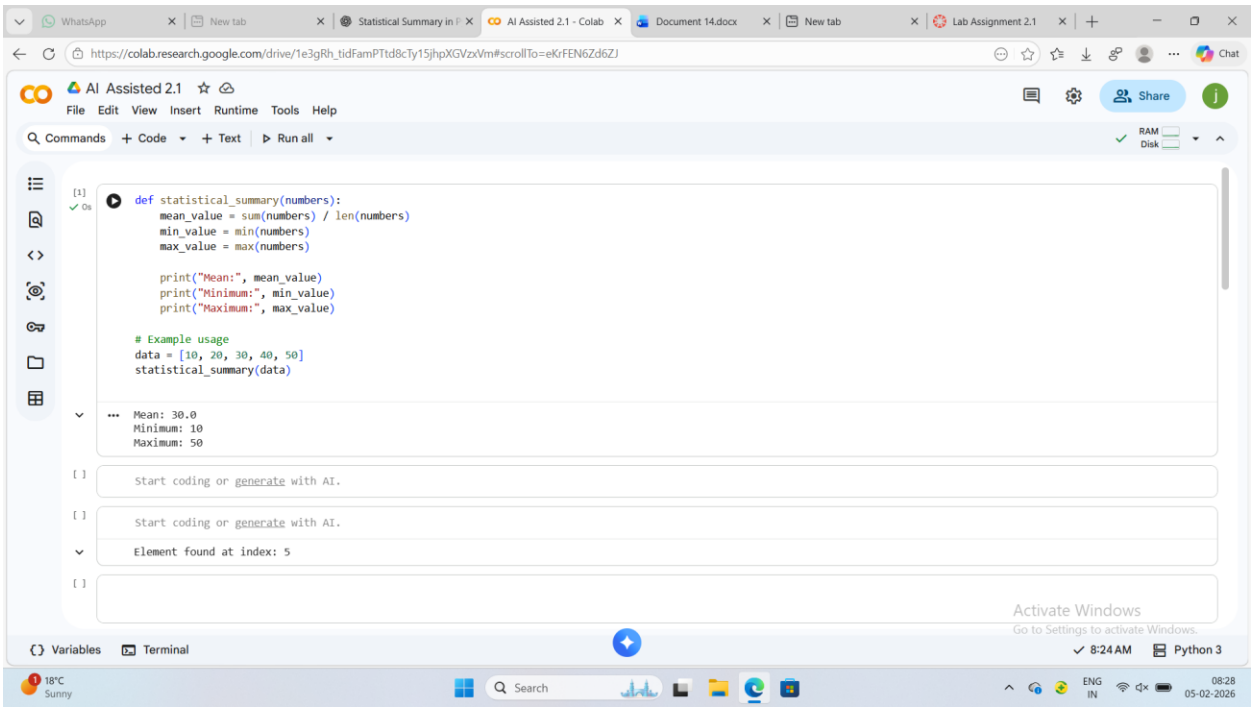
❖ Expected Output:

➢ Correct Python function

➢ Output shown in Colab

➢ Screenshot of Gemini prompt and result

Code



Explanation

In this task, survey responses are stored as a list of numbers. The goal is to analyze this data by calculating basic statistical values such as **mean**, **minimum**, and **maximum**.

A Python function named `statistical_summary` is created to perform this analysis.

How the program works:

1. **Function Definition**
   The function `statistical_summary(numbers)` accepts a list of numerical values as input.
2. **Mean Calculation**
   a. The `sum()` function adds all the values in the list.
   b. The total is divided by the number of elements using `len(numbers)` to calculate the mean.
3. **Minimum Value**
   a. The built-in `min()` function finds the smallest value in the list.
4. **Maximum Value**
   a. The built-in `max()` function finds the largest value in the list.
5. **Displaying Output**
   a. The calculated mean, minimum, and maximum values are printed so the result is visible in Google Colab.

Conclusion

This program helps summarize survey data quickly and accurately. It demonstrates the use of Python functions and built-in methods to perform basic statistical analysis efficiently.

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.

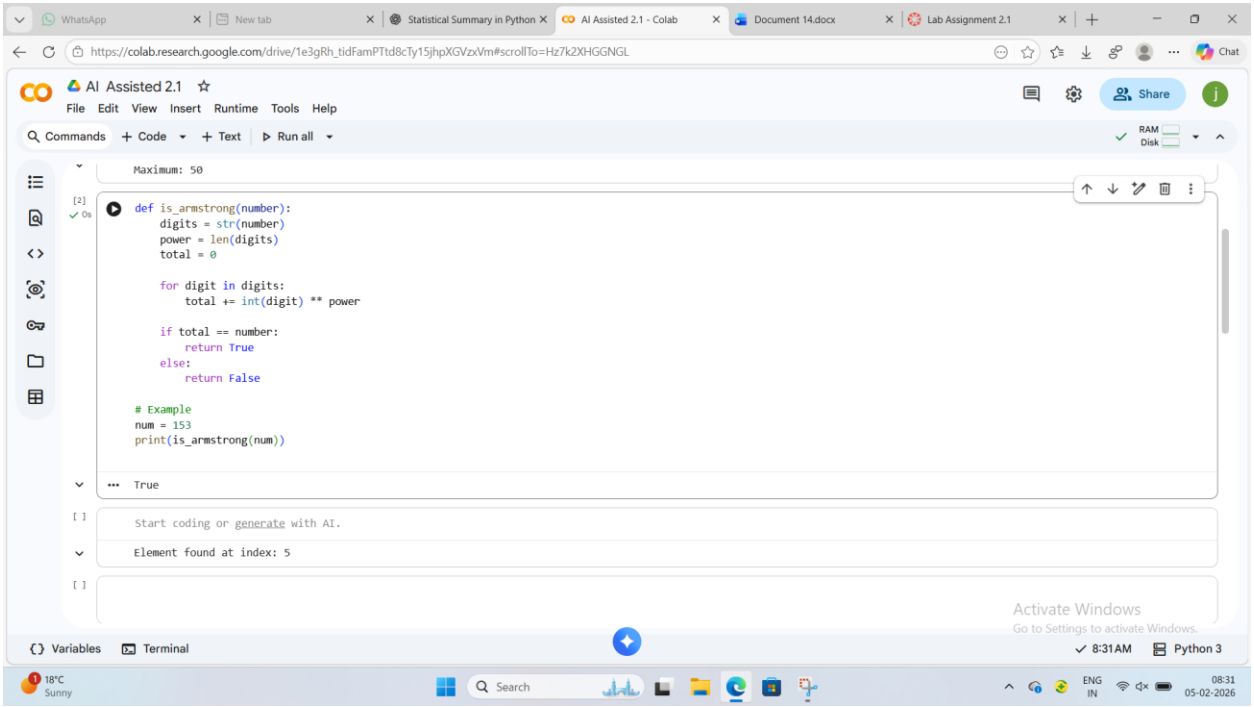
Compare their outputs, logic style, and clarity.

❖ Expected Output:

➢ Side-by-side comparison table

➢ Screenshots of prompts and generated code

Code



Explanation

An **Armstrong number** is a number that is equal to the sum of its digits raised to the power of the total number of digits. This concept is commonly used to test numeric validation logic in programming.

For example:
153 is an Armstrong number because
$13+53+33=1531^3 + 5^3 + 3^3 = 153$13+53+33=153

Gemini Approach Explanation

Gemini generates a **function-based** and **readable** solution.

- The number is first converted into a string to easily access each digit.
- The total number of digits is calculated using `len()`.
- Each digit is converted back to an integer and raised to the required power.
- The sum of powered digits is compared with the original number.
- The function returns `True` if the number is an Armstrong number, otherwise `False`.

Conclusion

Both AI tools successfully generate Armstrong number checkers with correct logic. However, Gemini provides more readable and beginner-friendly code, whereas GitHub Copilot produces compact and performance-oriented code. The comparison highlights how different AI tools approach the same problem in different styles.

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.

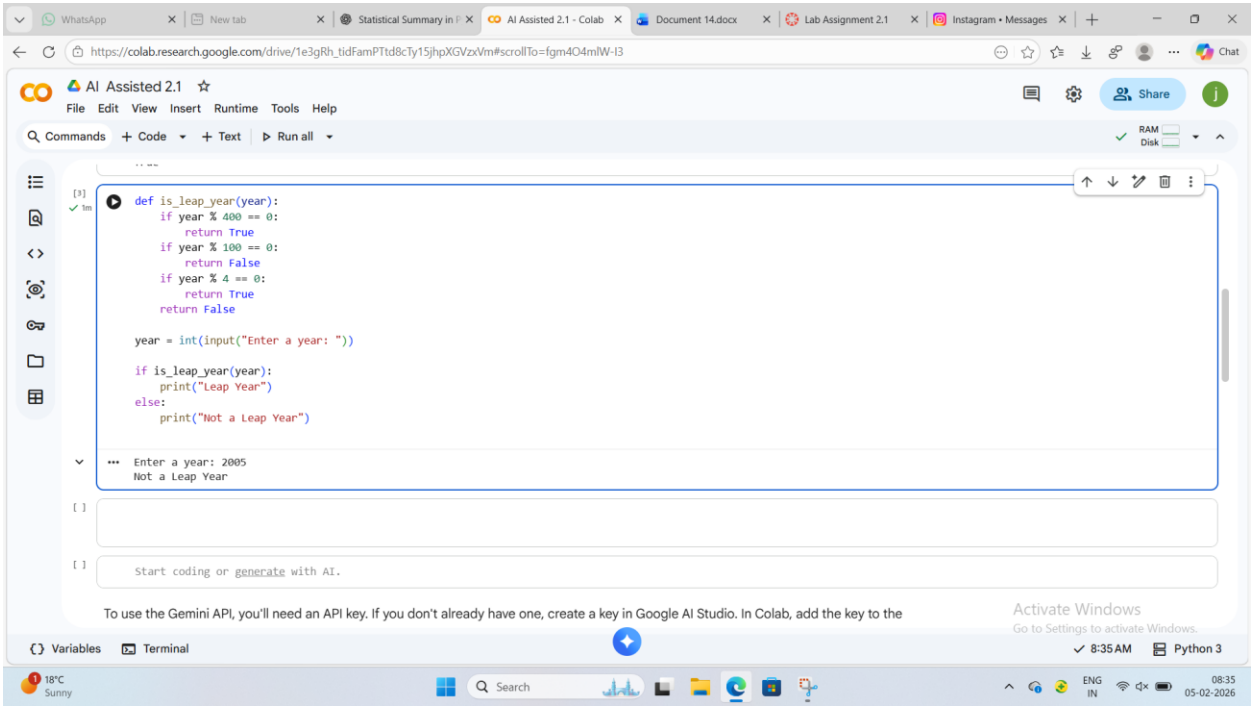
❖ Expected Output:

➢ Two versions of code

➢ Sample inputs/outputs

➢ Brief comparison

Code



Explanation

- **User Input**
- The program reads a year from the user using `input()`.
- `int()` converts the input from string to integer.
- **Leap Year Condition**
- `year % 400 == 0`: Checks if the year is divisible by 400.
- `year % 4 == 0 and year % 100 != 0`: Checks if the year is divisible by 4 but not by 100.
- The logical `or` operator ensures that either condition being true makes it a leap year.
- **Output**
- If the condition is true, the program prints **"Leap Year"**.
- Otherwise, it prints **"Not a Leap Year"**.

Conclusion

The leap year validation programs generated using Cursor AI correctly implement standard leap year rules. By using different prompts, Cursor AI produced different coding styles—one with direct conditional logic and another with a function-based approach. The simple prompt resulted in a concise and efficient program suitable for quick checks, while the detailed prompt produced modular and reusable code ideal for backend systems. This task demonstrates that prompt design significantly influences the structure, readability, and reusability of AI-generated code.

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.

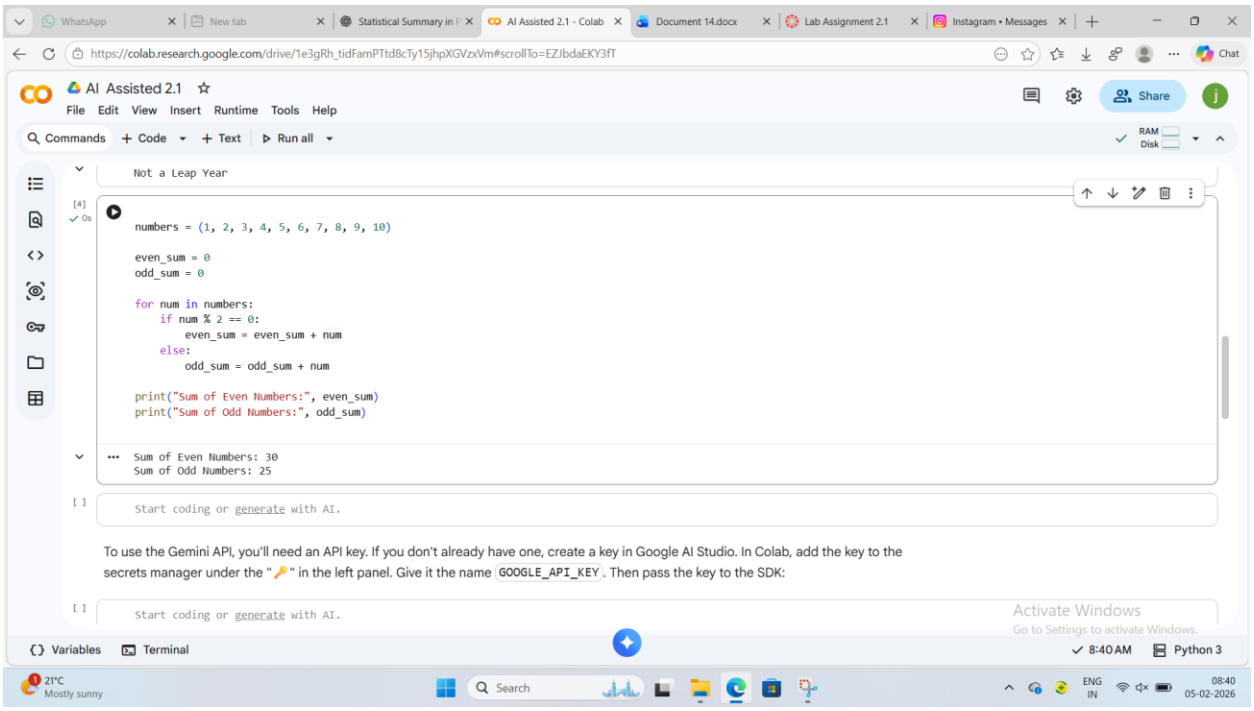
❖ Expected Output:

➢ Original code

➢ Refactored code

➢ Explanation of improvements

Code



Explanation

- **Code Length Reduced**
  The refactored version uses Python's built-in `sum()` function, reducing the number of lines.
- **Better Readability**
  Generator expressions clearly show the condition for even and odd numbers in a single line.
- **Efficiency**
  The logic is concise and avoids manual variable updates inside a loop.
- **Pythonic Style**
  The refactored code follows modern Python best practices, making it cleaner and easier to maintain.
- **Same Logic, Better Structure**
  The core logic written by the student remains unchanged; only the implementation is improved.

Conclusion

This task demonstrates how initial problem-solving logic written by a developer can be enhanced using AI tools. AI refactoring improves code clarity, efficiency, and maintainability without changing the original functionality.