AI Assisted Coding
LAB-02

Roll No : 2303A51863
Batch : 13
Name : K.Bhavya sri

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

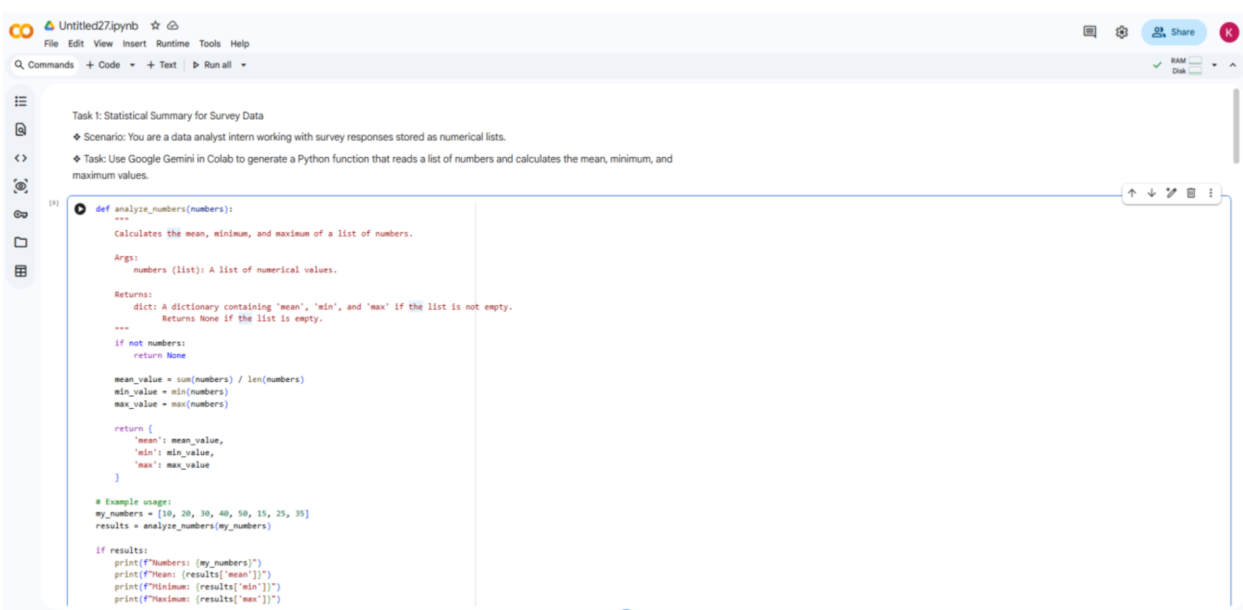
➢ Output shown in Colab

➢ Screenshot of Gemini prompt and result

<u>Prompt:</u>

Generate a Python function that accepts a list of numbers as a parameter and calculates the mean, minimum, and maximum values. The function should return or display the results clearly.

<u>Code&Output:</u>



```python
results = analyze_numbers(my_numbers)

if results:
    print(f"Numbers: {my_numbers}")
    print(f"Mean: {results['mean']}")
    print(f"Minimum: {results['min']}")
    print(f"Maximum: {results['max']}")
else:
    print("The list of numbers is empty.")

empty_list = []
empty_results = analyze_numbers(empty_list)
if empty_results:
    print(f"Mean: {empty_results['mean']}")
else:
    print(f"Numbers: {empty_list}")
    print("The empty list of numbers returned None as expected.")
```

```
Numbers: [10, 20, 30, 40, 50, 15, 25, 35]
Mean: 28.125
Minimum: 10
Maximum: 50
Numbers: []
The empty list of numbers returned None as expected.
```

Justification:

The function analyzes a list of survey numbers by calculating the mean, minimum, and maximum values using built-in Python functions. It also checks for empty input to avoid errors and displays the results clearly in Google Colab. This ensures accurate and efficient statistical analysis of the data.

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.

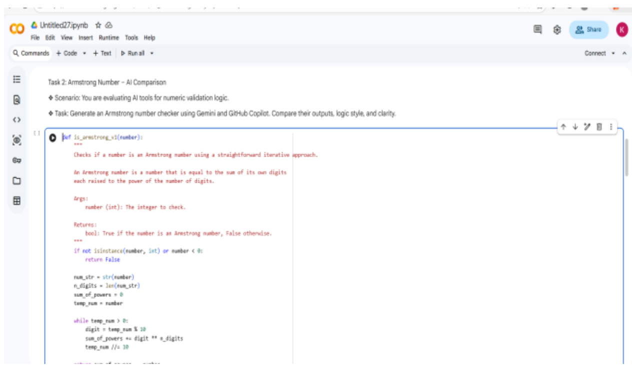
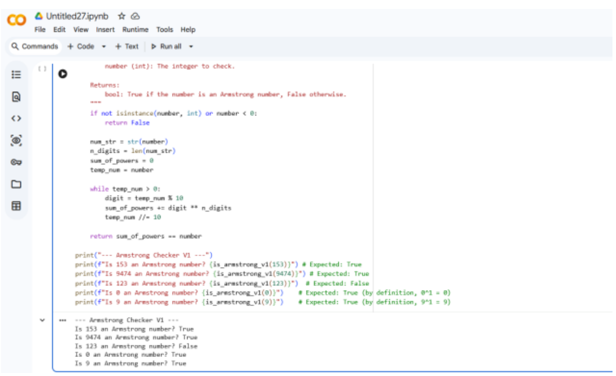
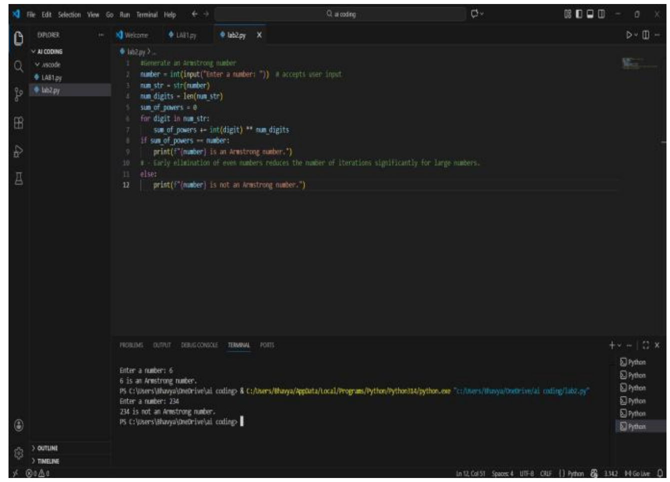
❖ Expected Output:

➢ Side-by-side comparison table

➢ Screenshots of prompts and generated code

Prompt:

Write a Python program to check whether a number is an Armstrong number.

Aspect	Google Colab	VS Code
Code	 	
Output	Tests multiple predefined numbers and prints **True / False** for each case.	Takes a single user input and prints whether it is Armstrong or not.
Logic Style	Step-by-step approach using loops, explicit variables, and manual summation.	Compact logic using Python's `sum()` with a generator expression.
Input Handling	No user input – uses hard-coded test values.	Accepts user input from the keyboard.

Error Handling	Handles negative and non-integer input with a message.	No explicit validation for negative or invalid input.
Readability	Very clear for beginners, easy to follow line by line.	Short and clean but slightly advanced for beginners.
Reusability	Function is reusable, but demo code is fixed.	Function is reusable and flexible with user input.

Justification:
The comparison shows how different AI tools generate solutions for the same problem. Google Colab (Gemini) provides step-by-step logic that is easy for beginners to understand and includes basic validation. VS Code (GitHub Copilot) generates concise and efficient code suitable for faster development. This highlights the difference between learning-oriented and productivity-oriented AI tools.

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:

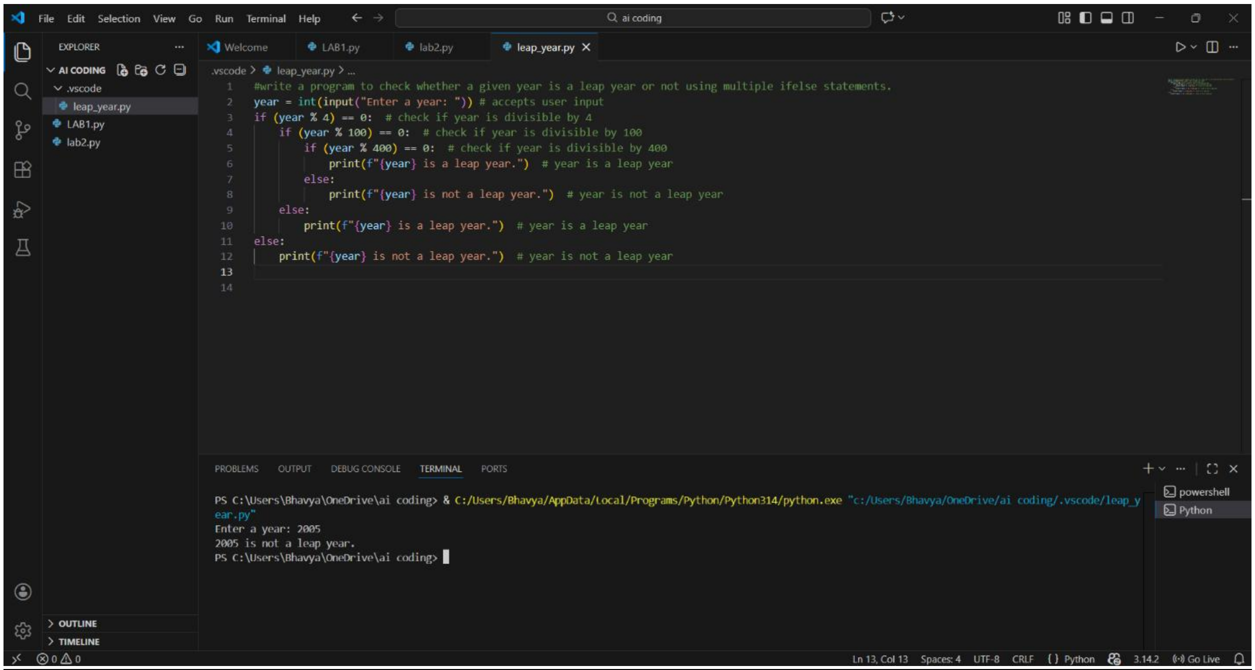
➢ Two versions of code

➢ Sample inputs/outputs

➢ Brief comparison

Prompt (1):
Generate a Python program that checks whether a given year is a leap year.

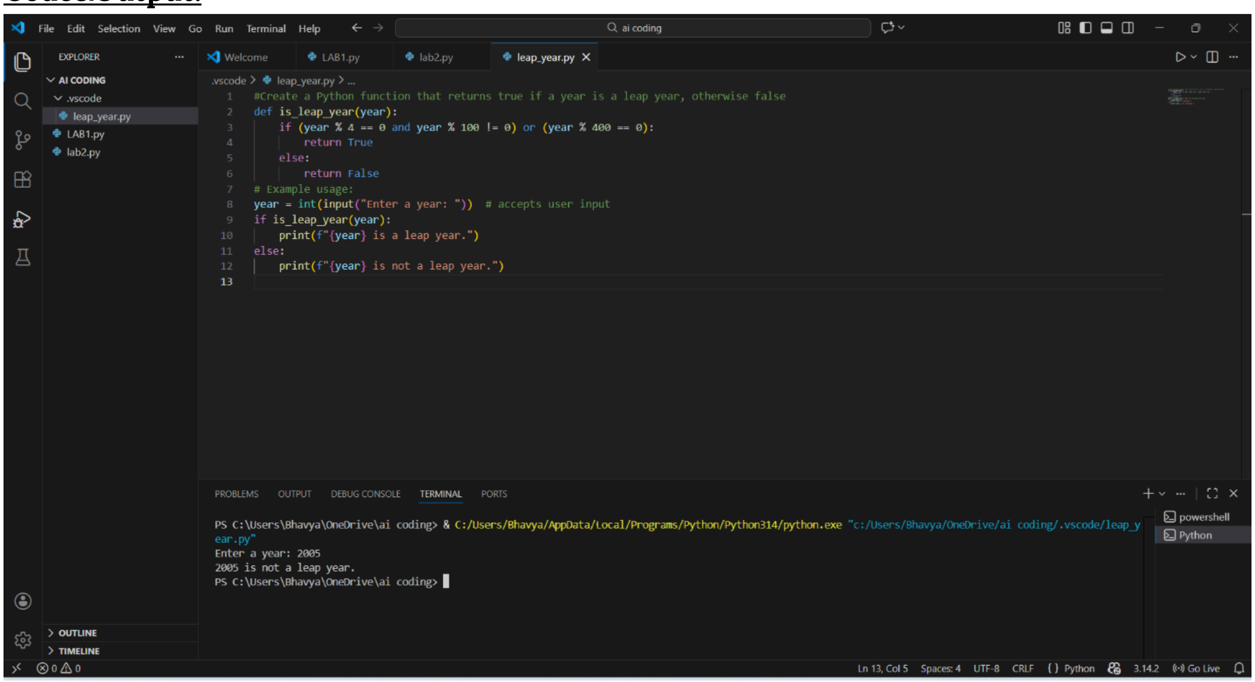
Code&Output:



Prompt (2):

Generate a Python program to check whether a given year is a leap year. Simplify the logic using a single conditional expression instead of nested if statements.

Code&Output:



Comparison of Two Leap Year Programs:

Feature	Prompt(1) – Nested If Method	Prompt(2) - Single Condition Method
Logic Style	Uses multiple nested `if-else` blocks	Uses one boolean expression
Lines of Code	More lines	Fewer lines
Readability	Harder to read due to nesting	Clean and easy to understand
Performance	Slightly slower due to multiple checks	Faster due to single evaluation
Maintainability	More complex to modify	Easy to modify
Pythonic Style	Traditional logic	Pythonic and optimized

Justification:
Two different prompts were given to Cursor AI to generate leap year validation code, resulting in different coding styles. The first version uses nested if-else statements, making the logic clear and easy to understand step by step. The second version simplifies the same logic into a single conditional expression, reducing code length and improving readability. Both versions produce the same correct output for all inputs. This demonstrates how prompt variation can influence code structure and optimization.

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

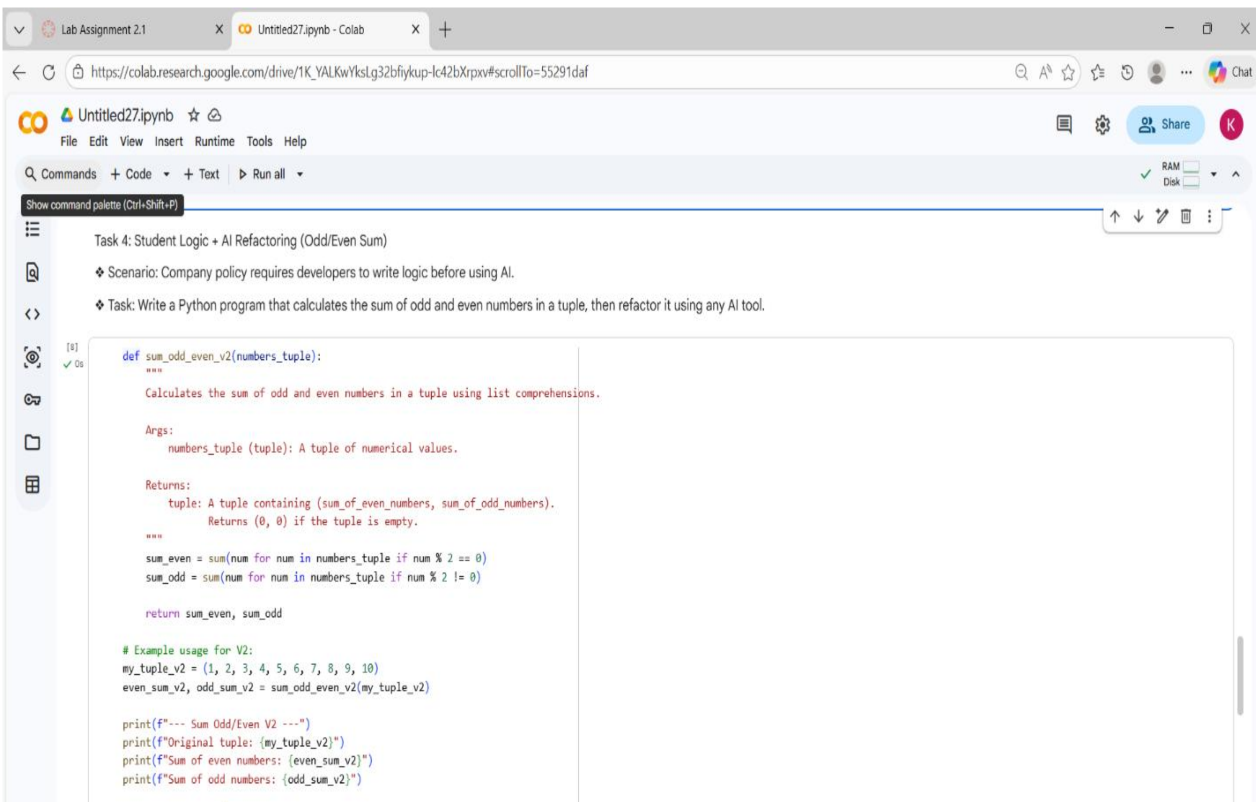
➢ Explanation of improvements

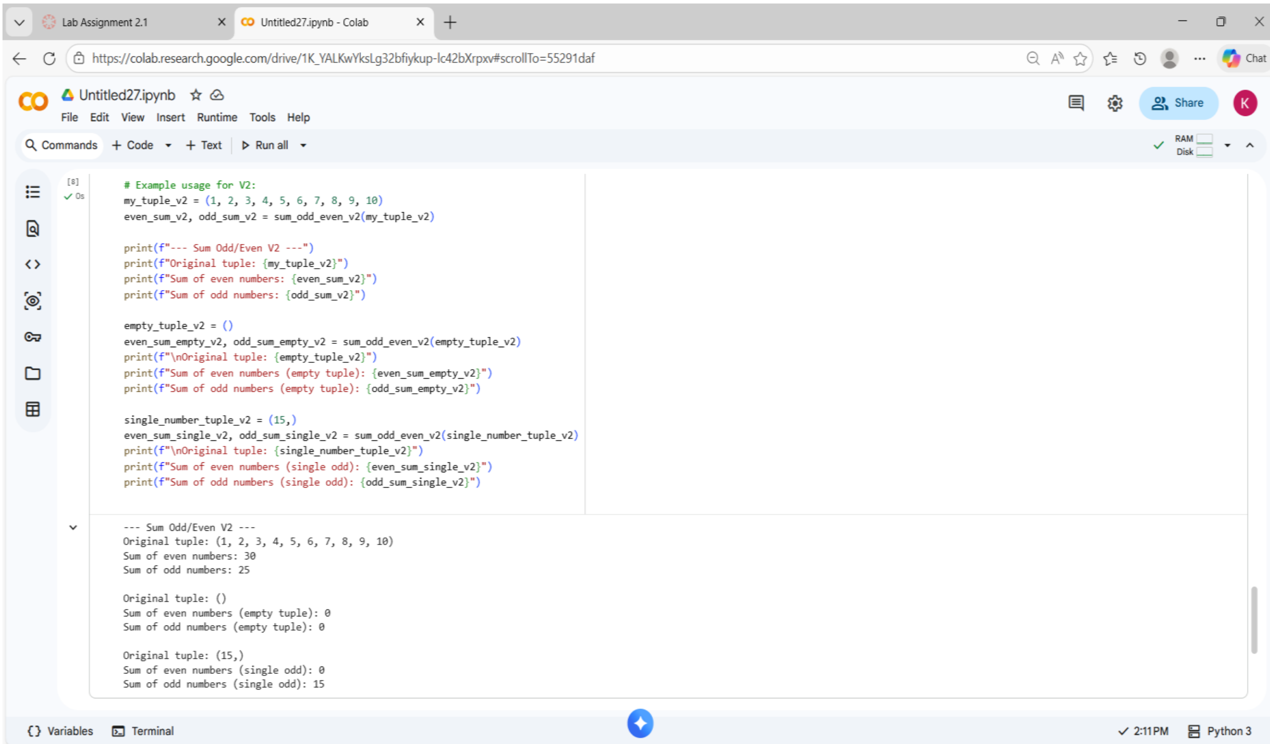
#Python program that calculates the sum of odd and even numbers in a tuple(My Code)

<u>Prompt:</u>

Python program that calculates the sum of odd and even numbers in a tuple(Refactor Code).

<u>Code:</u>





Improvement Comparison Table (Using GitHub Copilot):

Aspect	Original Code	Refactored Code
Logic Style	Uses loop and manual addition with variables.	Uses `sum()` with generator expressions.
Code Length	More lines and repetitive statements	Shorter and more compact code.
Readability & Efficiency	Clear but slightly lengthy and slower due to manual processing.	Cleaner, easier to read, and more efficient using built-in functions.

Justification:

Using GitHub Copilot, the original loop-based program was refactored by applying Python's built-in `sum()` function with generator expressions. This reduced the number of lines and eliminated unnecessary variables, making the code cleaner and easier to maintain. The logic became more readable and efficient while producing the same output. This demonstrates how AI tools can improve code quality without changing functionality.