

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.

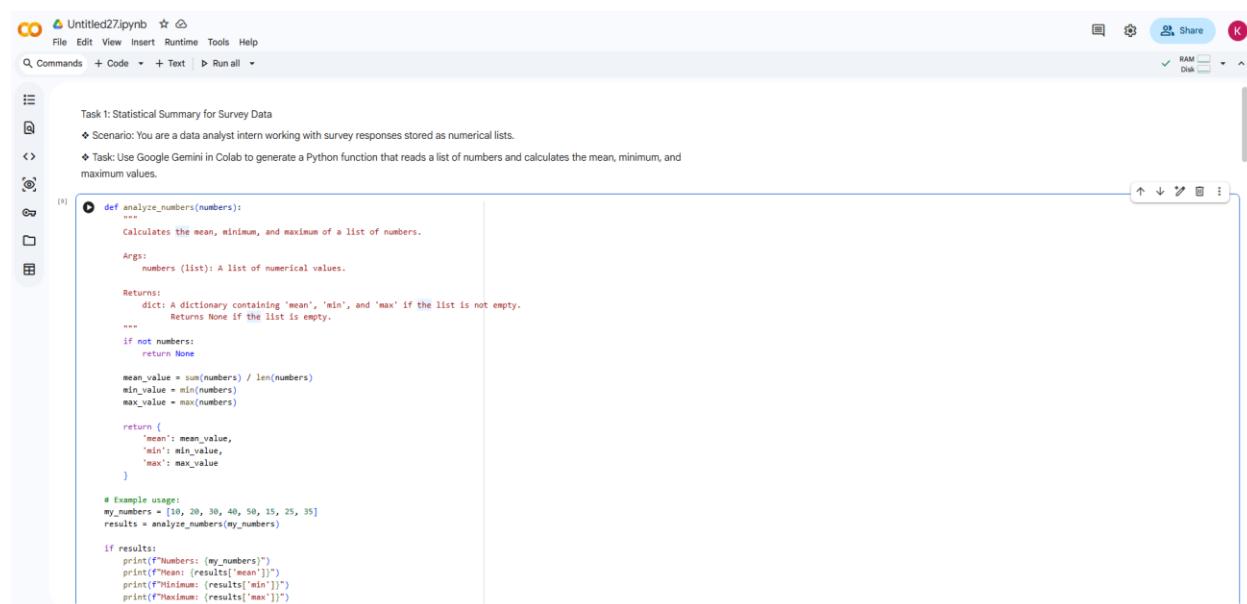
❖ Expected Output:

- Correct Python function
- Output shown in Colab
- Screenshot of Gemini prompt and result

Prompt:

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.

Code&Output:



```
File Edit View Insert Runtime Tools Help
Q Commands + Code + Text | Run all
Share RAM Disk
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.

def analyze_numbers(numbers):
    """
    Calculates the mean, minimum, and maximum of a list of numbers.

    Args:
        numbers (list): A list of numerical values.

    Returns:
        dict: A dictionary containing 'mean', 'min', and 'max' if the list is not empty.
        None: Returns None if the list is empty.
    """
    if not numbers:
        return None

    mean_value = sum(numbers) / len(numbers)
    min_value = min(numbers)
    max_value = max(numbers)

    return {
        'mean': mean_value,
        'min': min_value,
        'max': max_value
    }

# Example usage:
my_numbers = [10, 20, 30, 40, 50, 15, 25, 35]
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']}")
```

The screenshot shows a Google Colab notebook cell. The code defines a function `analyze_numbers` that takes a list of numbers as input. It prints the mean, minimum, and maximum values if the list is not empty; otherwise, it prints a message stating that the list is empty. A second part of the code creates an empty list, calls the function with it, and then prints the results, which correctly state that the empty list returned None as expected.

```
results = analyze_numbers(my_numbers)

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

empty_list = []
empty_results = analyze_numbers(empty_list)
if empty_results:
    print("Mean: (empty_results['mean'])")
else:
    print("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.

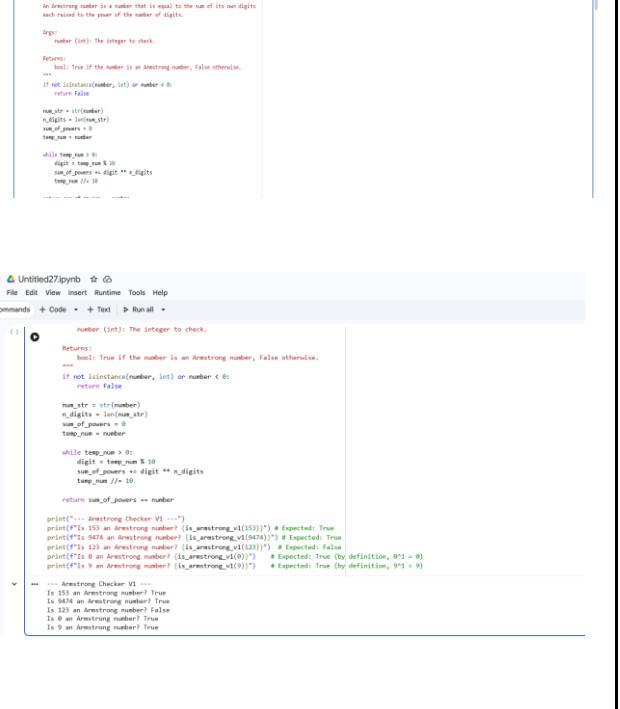
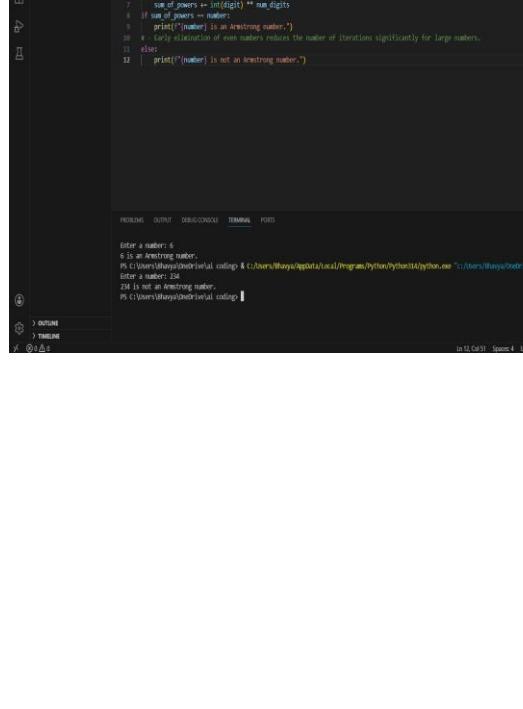
❖ Expected Output:

➢ 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	 <pre> def is_armstrong_v1(number): """ Checks if a number is an Armstrong number using a straightforward iterative approach. An Armstrong number is a number that is equal to the sum of its own digits each raised to the power of the number of digits. Args: number (int): The integer to check. Returns: bool: True if the number is an Armstrong number, False otherwise. """ if not isinstance(number, int) or number < 0: return False num_str = str(number) n_digits = len(num_str) sum_of_powers = 0 temp_num = number while temp_num > 0: digit = temp_num % 10 sum_of_powers += digit ** n_digits temp_num /= 10 return sum_of_powers == number def armstrong_checker_v1(): print("... Armstrong Checker V1 ...") print(f"Is 153 an Armstrong number? {is_armstrong_v1(153)}") # Expected: True print(f"Is 370 an Armstrong number? {is_armstrong_v1(370)}") # Expected: True print(f"Is 135 an Armstrong number? {is_armstrong_v1(135)}") # Expected: True print(f"Is 9474 an Armstrong number? {is_armstrong_v1(9474)}") # Expected: True print(f"Is 9 an Armstrong number? {is_armstrong_v1(9)}") # Expected: True print(f"Is 0 an Armstrong number? {is_armstrong_v1(0)}") # Expected: True print(f"Is -9 an Armstrong number? {is_armstrong_v1(-9)}") # Expected: False print(f"Is 123 an Armstrong number? {is_armstrong_v1(123)}") # Expected: False print(f"Is 8 an Armstrong number? {is_armstrong_v1(8)}") # Expected: True (By definition, 8**1 = 8) print(f"Is 999 an Armstrong number? {is_armstrong_v1(999)}") # Expected: True </pre>	 <pre> def is_armstrong_v1(number): """ Checks if a number is an Armstrong number using a straightforward iterative approach. An Armstrong number is a number that is equal to the sum of its own digits each raised to the power of the number of digits. Args: number (int): The integer to check. Returns: bool: True if the number is an Armstrong number, False otherwise. """ if not isinstance(number, int) or number < 0: return False num_str = str(number) n_digits = len(num_str) sum_of_powers = 0 temp_num = number while temp_num > 0: digit = temp_num % 10 sum_of_powers += digit ** n_digits temp_num /= 10 return sum_of_powers == number if __name__ == "__main__": print(f"Is 153 an Armstrong number? {is_armstrong_v1(153)}") print(f"Is 370 an Armstrong number? {is_armstrong_v1(370)}") print(f"Is 135 an Armstrong number? {is_armstrong_v1(135)}") print(f"Is 9474 an Armstrong number? {is_armstrong_v1(9474)}") print(f"Is 9 an Armstrong number? {is_armstrong_v1(9)}") print(f"Is 0 an Armstrong number? {is_armstrong_v1(0)}") print(f"Is -9 an Armstrong number? {is_armstrong_v1(-9)}") print(f"Is 123 an Armstrong number? {is_armstrong_v1(123)}") print(f"Is 8 an Armstrong number? {is_armstrong_v1(8)}") print(f"Is 999 an Armstrong number? {is_armstrong_v1(999)}") </pre>
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 <code>sum()</code> 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:

A screenshot of the Visual Studio Code (VS Code) interface. The left sidebar shows a tree view with 'EXPLORER' expanded, containing 'AI CODING' and '.vscode'. Under 'AI CODING', there are files 'leap.year.py', 'LAB1.py', and 'lab2.py'. The main editor area displays a Python script named 'leap.year.py'. The code checks if a given year is a leap year using multiple if-else statements. The terminal at the bottom shows the output of running the script with the input '2005', which correctly identifies it as not being a leap year.

```
.vscode > leap.year.py > ...
1 #write a program to check whether a given year is a leap year or not using multiple ifelse statements.
2 year = int(input("Enter a year: ")) # accepts user input
3 if (year % 4) == 0: # check if year is divisible by 4
4     if (year % 100) == 0: # check if year is divisible by 100
5         if (year % 400) == 0: # check if year is divisible by 400
6             print(f"{year} is a leap year.") # year is a leap year
7         else:
8             print(f"{year} is not a leap year.") # year is not a leap year
9     else:
10        print(f"{year} is a leap year.") # year is a leap year
11 else:
12     print(f"{year} is not a leap year.") # year is not a leap year
13
14
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Bhavya\OneDrive\ai coding & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe "c:/Users/Bhavya/OneDrive/ai coding/.vscode/leap.year.py"
Enter a year: 2005
2005 is not a leap year.
PS C:\Users\Bhavya\OneDrive\ai coding>
Ln 13, Col 13  Spaces: 4  UTF-8  CRLF  {} Python  3.14.2  ⓘ Go Live
```

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:

A screenshot of the Visual Studio Code (VS Code) interface. The left sidebar shows a tree view with 'EXPLORER' expanded, containing 'AI CODING' and '.vscode'. Under 'AI CODING', there are files 'leap.year.py', 'LAB1.py', and 'lab2.py'. The main editor area displays a Python script named 'leap.year.py'. The code defines a function 'is_leap_year' that returns True if a year is a leap year (i.e., if it is divisible by 4 but not by 100, or if it is divisible by 400). It then uses this function to check if the input year '2005' is a leap year. The terminal at the bottom shows the output of running the script with the input '2005', which correctly identifies it as not being a leap year.

```
.vscode > leap.year.py > ...
1 #Create a Python function that returns true if a year is a leap year, otherwise false
2 def is_leap_year(year):
3     if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
4         return True
5     else:
6         return False
7 # Example usage:
8 year = int(input("Enter a year: ")) # accepts user input
9 if is_leap_year(year):
10    print(f"{year} is a leap year.")
11 else:
12    print(f"{year} is not a leap year.")
13
14
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Bhavya\OneDrive\ai coding & C:/Users/Bhavya/AppData/Local/Programs/Python/Python314/python.exe "c:/Users/Bhavya/OneDrive/ai coding/.vscode/leap.year.py"
Enter a year: 2005
2005 is not a leap year.
PS C:\Users\Bhavya\OneDrive\ai coding>
Ln 13, Col 5  Spaces: 4  UTF-8  CRLF  {} Python  3.14.2  ⓘ Go Live
```

Comparison of Two Leap Year Programs:

Feature	Prompt(1) - Nested If Method	Prompt(2) - Single Condition Method
Logic Style	Uses multiple nested <code>if-else</code> 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:

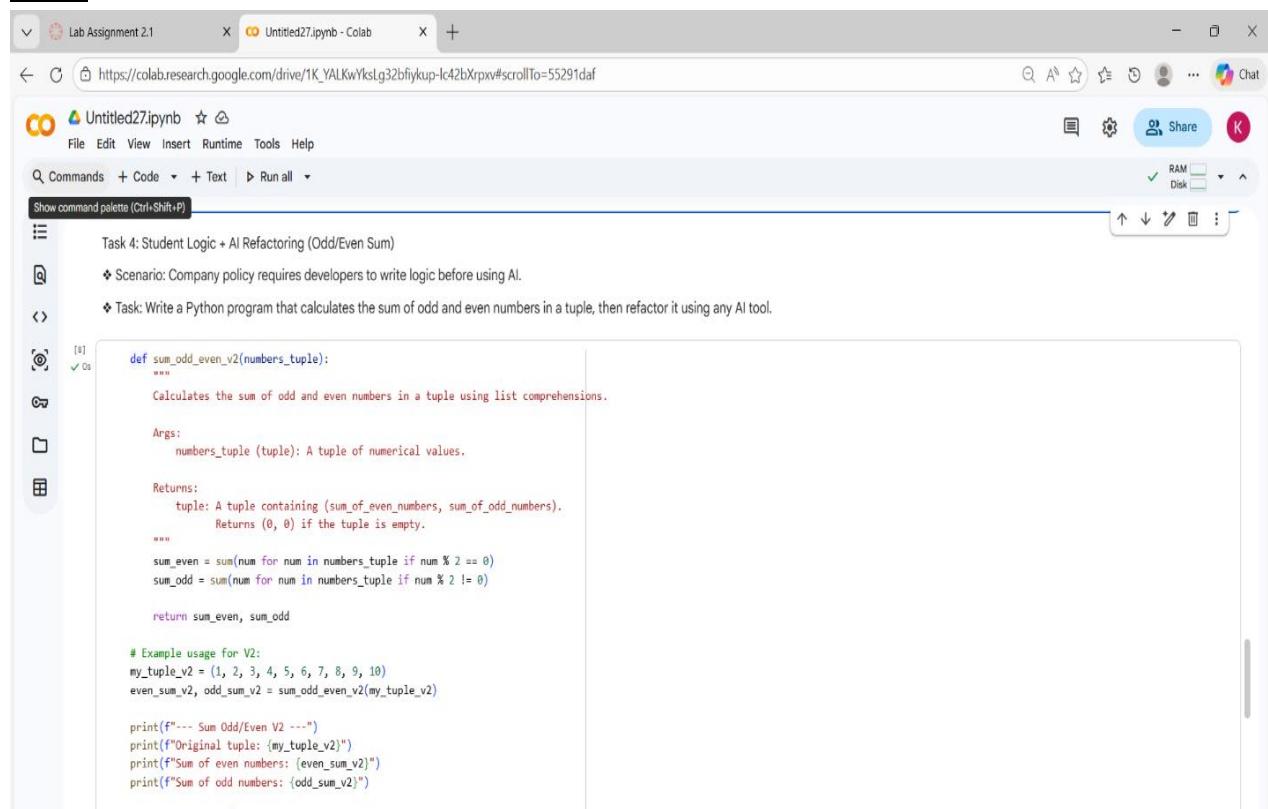
- Original code
- Refactored code
- Explanation of improvements

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

Prompt:

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

Code:



The screenshot shows a Google Colab notebook titled "Untitled27.ipynb". The code cell contains a function definition for calculating the sum of odd and even numbers in a tuple using list comprehensions. The AI refactoring sidebar on the right provides context about the task and suggests improvements.

```
def sum_odd_even_v2(numbers_tuple):
    """
    Calculates the sum of odd and even numbers in a tuple using list comprehensions.

    Args:
        numbers_tuple (tuple): A tuple of numerical values.

    Returns:
        tuple: A tuple containing (sum_of_even_numbers, sum_of_odd_numbers).
        Returns (0, 0) if the tuple is empty.

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

    return sum_even, sum_odd

# Example usage for V2:
my_tuple_v2 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
even_sum_v2, odd_sum_v2 = sum_odd_even_v2(my_tuple_v2)

print("---- Sum Odd/Even V2 ----")
print(f"Original tuple: {my_tuple_v2}")
print(f"Sum of even numbers: {even_sum_v2}")
print(f"Sum of odd numbers: {odd_sum_v2}")
```

```

# Example usage for V2:
my_tuple_v2 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
even_sum_v2, odd_sum_v2 = sum_odd_even_v2(my_tuple_v2)

print("---- Sum Odd/Even V2 ----")
print("Original tuple: (my_tuple_v2)")
print(f"Sum of even numbers: {even_sum_v2}")
print(f"Sum of odd numbers: {odd_sum_v2}")

empty_tuple_v2 = ()
even_sum_empty_v2, odd_sum_empty_v2 = sum_odd_even_v2(empty_tuple_v2)
print("\nOriginal tuple: (empty_tuple_v2)")
print(f"Sum of even numbers (empty tuple): {even_sum_empty_v2}")
print(f"Sum of odd numbers (empty tuple): {odd_sum_empty_v2}")

single_number_tuple_v2 = (15,)
even_sum_single_v2, odd_sum_single_v2 = sum_odd_even_v2(single_number_tuple_v2)
print("\nOriginal tuple: (single_number_tuple_v2)")
print(f"Sum of even numbers (single odd): {even_sum_single_v2}")
print(f"Sum of odd numbers (single odd): {odd_sum_single_v2}")

--- Sum Odd/Even V2 ---
Original tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Sum of even numbers: 30
Sum of odd numbers: 25

Original tuple: ()
Sum of even numbers (empty tuple): 0
Sum of odd numbers (empty tuple): 0

Original tuple: (15,)
Sum of even numbers (single odd): 0
Sum of odd numbers (single odd): 15

```

Improvement Comparison Table (Using GitHub Copilot):

Aspect	Original Code	Refactored Code
Logic Style	Uses loop and manual addition with variables.	Uses <code>sum()</code> 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.