

## Lab Assignment – 2

Name : L .Srinivas Reddy

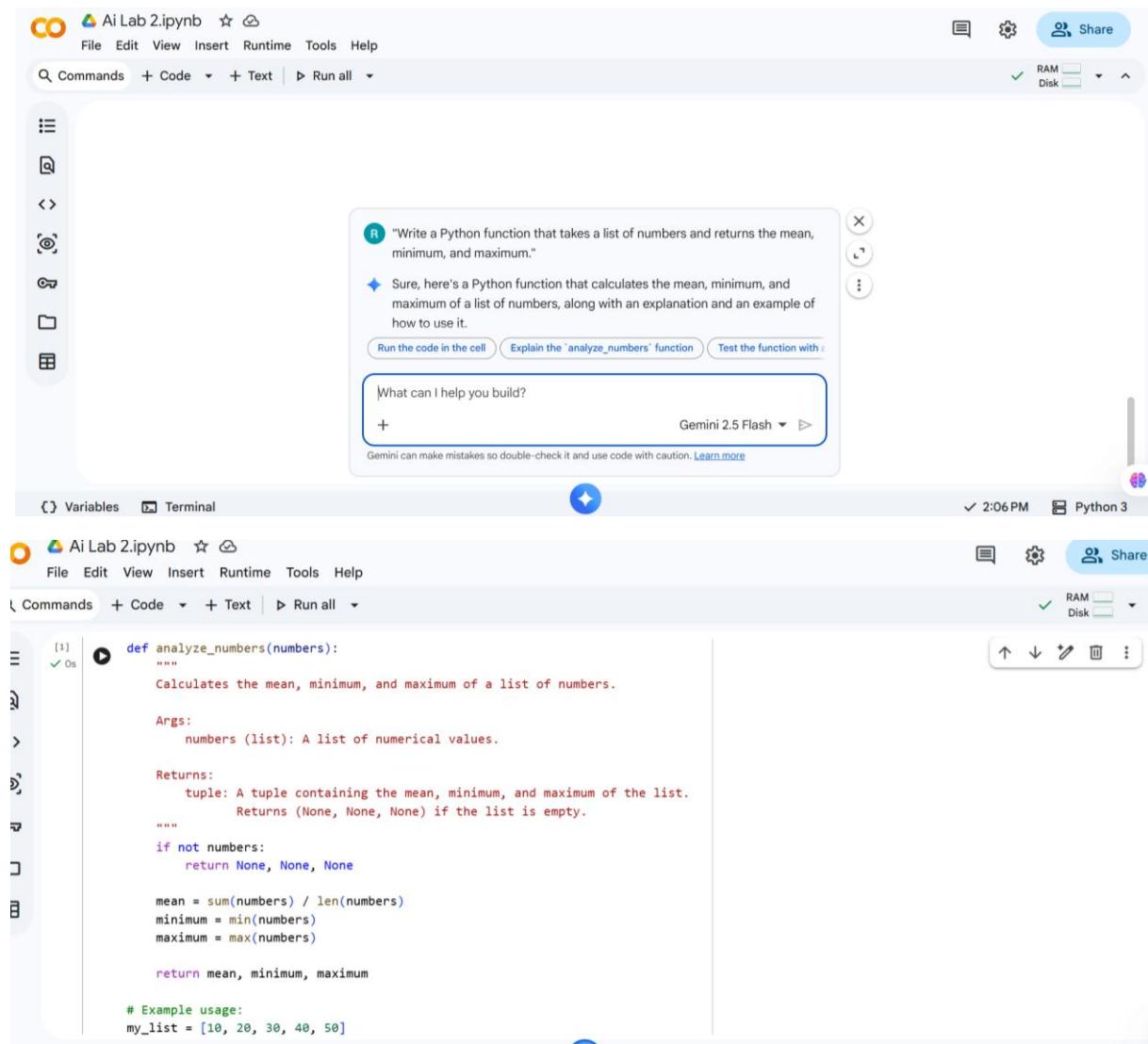
HT.No : 2303A52274

Batch : 36

### Task 1: Statistical Summary for Survey Data

Use Google Gemini in Colab to generate a Python function that reads a list of numbers and calculates the **mean, minimum, and maximum** values.

#### 1 . Screenshot of Gemini prompt and result



The screenshot shows the Google Colab interface with two main sections. The top section displays a Gemini AI prompt asking for a Python function to calculate mean, minimum, and maximum from a list of numbers. Below the prompt is a text input field and a 'Gemini 2.5 Flash' dropdown. The bottom section shows the generated Python code in a code cell:

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

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

    Returns:
        tuple: A tuple containing the mean, minimum, and maximum of the list.
        Returns (None, None, None) if the list is empty.
    """
    if not numbers:
        return None, None, None

    mean = sum(numbers) / len(numbers)
    minimum = min(numbers)
    maximum = max(numbers)

    return mean, minimum, maximum

# Example usage:
my_list = [10, 20, 30, 40, 50]
```

A screenshot of a Jupyter Notebook interface. The title bar says "Ai Lab 2.ipynb". The code cell contains Python code for a function named "analyze\_numbers". The code includes examples for both non-empty and empty lists.

```
[1]  ✓ 0s
minimum = min(numbers)
maximum = max(numbers)

return mean, minimum, maximum

# Example usage:
my_list = [10, 20, 30, 40, 50]
mean_val, min_val, max_val = analyze_numbers(my_list)

print(f"List: {my_list}")
print(f"Mean: {mean_val}")
print(f"Minimum: {min_val}")
print(f"Maximum: {max_val}")

# Example with an empty list
empty_list = []
mean_empty, min_empty, max_empty = analyze_numbers(empty_list)
print(f"\nEmpty List: {empty_list}")
print(f"Mean (empty): {mean_empty}")
print(f"Minimum (empty): {min_empty}")
print(f"Maximum (empty): {max_empty}")
```

## OUTPUT :

A screenshot of the Jupyter Notebook interface showing the output of the previous code cell. It displays the results for both the non-empty list and the empty list.

```
print(f"Maximum (empty): {max_empty}")
...
... List: [10, 20, 30, 40, 50]
Mean: 30.0
Minimum: 10
Maximum: 50

Empty List: []
Mean (empty): None
Minimum (empty): None
Maximum (empty): None
```

## Explanation

- **def analyze\_numbers(numbers)::** This defines a function named `analyze_numbers` that takes one argument, `numbers`, which is expected to be a list.
- **Docstring:** The triple-quoted string explains what the function does, its arguments (Args), and what it returns (Returns). This is good practice for documenting your code.
- **if not numbers::** This line checks if the input list `numbers` is empty. If it is, the function returns (None, None, None) to avoid errors (like division by zero for the mean or `min()/max()` on an empty list).
- **mean = sum(numbers) / len(numbers):** Calculates the mean (average) by summing all numbers in the list using `sum()` and dividing by the count of numbers using `len()`.
- **minimum = min(numbers):** Finds the smallest number in the list using the built-in `min()` function.
- **maximum = max(numbers):** Finds the largest number in the list using the built-in `max()` function.

- **return mean, minimum, maximum:** The function returns these three calculated values as a tuple.

## Task 2: Armstrong Number – AI Comparison

Generate an Armstrong number checker using Gemini and GitHub Copilot.

The screenshot shows the AI Lab 2.ipynb notebook interface. A pop-up window from Gemini 2.5 Flash displays a question: "Write a Python program to check whether a given number is an Armstrong number." Below the question, a response is provided: "Certainly! Here's a Python program that checks if a given number is an Armstrong number, along with an explanation of the code." At the bottom of the pop-up, there are buttons for "Accept & Run", "Accept", and "Cancel". The main notebook area is visible at the bottom.

The screenshot shows the AI Lab 2.ipynb notebook interface. The code cell contains a Python function named `is_armstrong`. The function checks if a given number is an Armstrong number by summing the digits raised to the power of the number of digits. It includes error handling for non-negative integers and examples of Armstrong numbers like 153, 370, and 371.

```

def is_armstrong(number):
    """
    Checks if a given number is an Armstrong number.

    An Armstrong number (also known as a narcissistic number, pluperfect digital invariant, or plus perfect number)
    is a number that is the sum of its own digits each raised
    to the power of the number of digits.

    For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153.

    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:
        print("Please enter a non-negative integer.")
        return False

    # Convert number to string to easily count digits and access individual digits
    num_str = str(number)

```

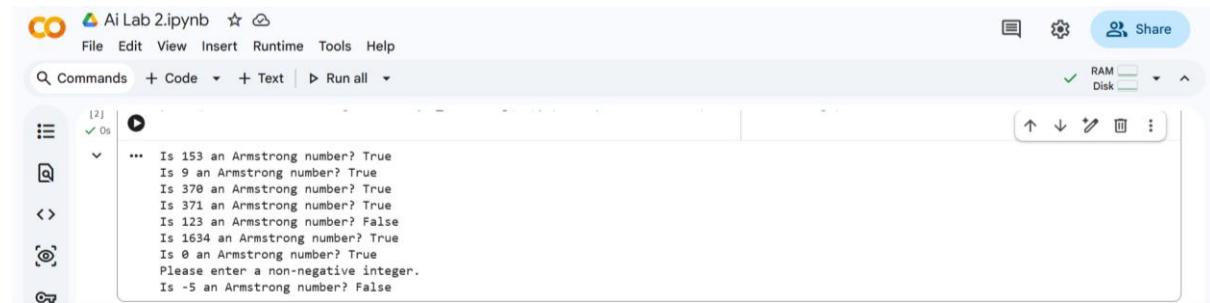
The screenshot shows the AI Lab 2.ipynb notebook interface. The code cell now includes a series of test cases using the `print` statement to verify the correctness of the `is_armstrong` function. The test cases cover various numbers, including positive integers, zero, and negative numbers.

```

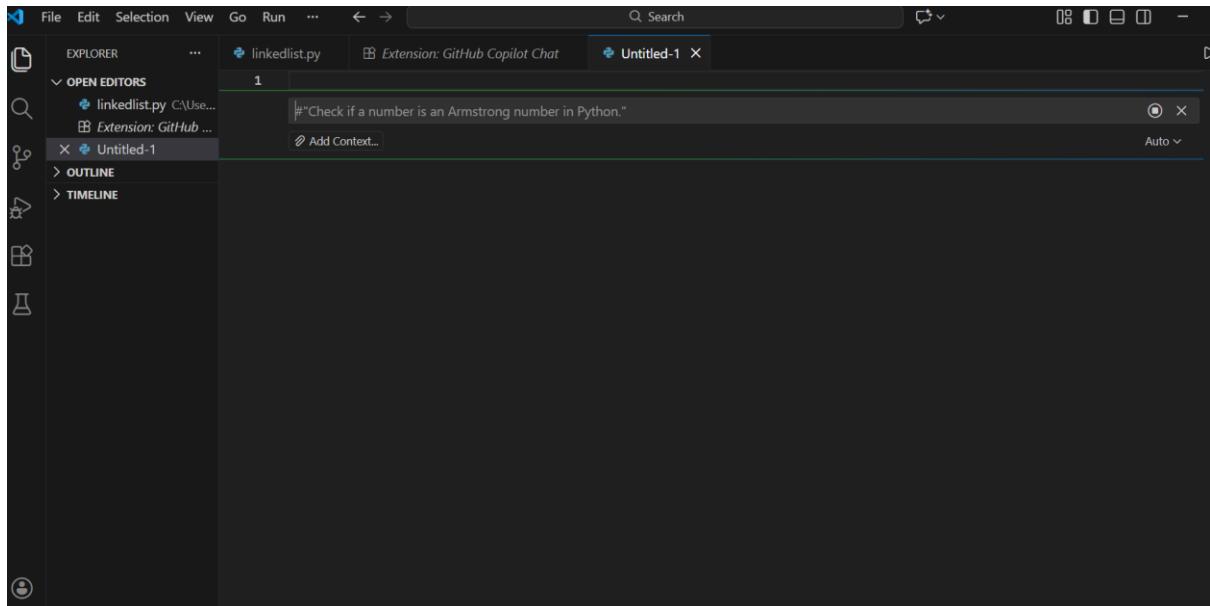
# Example usage:
print("Is 153 an Armstrong number? {is_armstrong(153)}") # Expected: True
print("Is 9 an Armstrong number? {is_armstrong(9)}") # Expected: True (1-digit numbers are Armstrong if digit^1 = digit)
print("Is 370 an Armstrong number? {is_armstrong(370)}") # Expected: True
print("Is 371 an Armstrong number? {is_armstrong(371)}") # Expected: True
print("Is 123 an Armstrong number? {is_armstrong(123)}") # Expected: False
print("Is 1634 an Armstrong number? {is_armstrong(1634)}") # Expected: True
print("Is 0 an Armstrong number? {is_armstrong(0)}") # Expected: True
print("Is -5 an Armstrong number? {is_armstrong(-5)}") # Expected: False (with error message)

```

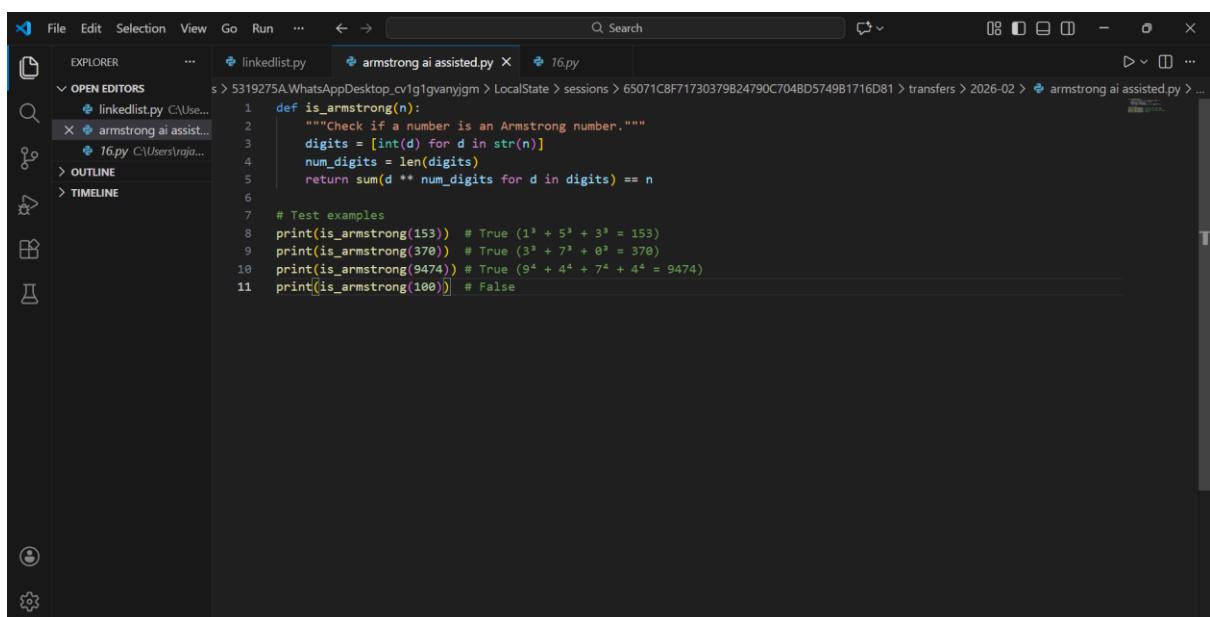
## OUTPUT :



```
... Is 153 an Armstrong number? True
Is 9 an Armstrong number? True
Is 370 an Armstrong number? True
Is 371 an Armstrong number? True
Is 123 an Armstrong number? False
Is 1634 an Armstrong number? True
Is 0 an Armstrong number? True
Please enter a non-negative integer.
Is -5 an Armstrong number? False
```



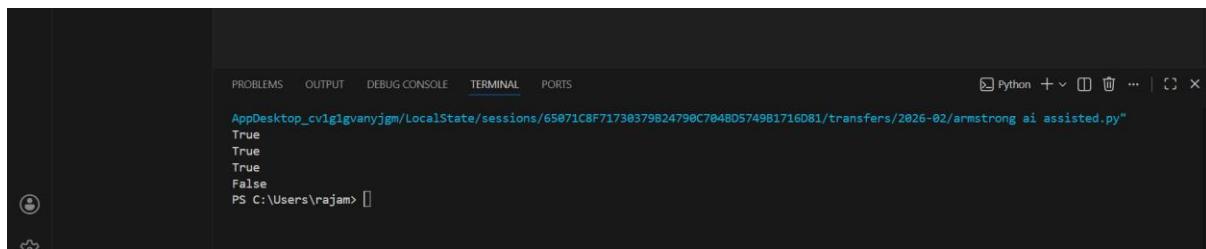
```
#Check if a number is an Armstrong number in Python.
```



```
def is_armstrong(n):
    """Check if a number is an Armstrong number."""
    digits = [int(d) for d in str(n)]
    num_digits = len(digits)
    return sum(d ** num_digits for d in digits) == n

# Test examples
print(is_armstrong(153)) # True (1³ + 5³ + 3³ = 153)
print(is_armstrong(370)) # True (3³ + 7³ + 0³ = 370)
print(is_armstrong(9474)) # True (9⁴ + 4⁴ + 7⁴ + 4⁴ = 9474)
print(is_armstrong(180)) # False
```

## OUTPUT :



A screenshot of a terminal window titled "Python". The window shows the command "PS C:\Users\rajam> []" at the bottom. Above it, there is a block of code and its output:

```
AppDesktop_cv1g1gvanyjgm\LocalState\sessions/65071C8F71730379B24790C704BD5749B1716D81/transfers/2026-02/armstrong ai assisted.py"
True
True
True
False
PS C:\Users\rajam> []
```

## Explanation :

- **def is\_armstrong(number)::** This defines a function named `is_armstrong` that takes one argument, `number`, which is expected to be an integer.
- **Docstring:** The triple-quoted string explains what the function does, its arguments (Args), and what it returns (Returns). It also clarifies the definition of an Armstrong number.
- **Input Validation:** if not `isinstance(number, int)` or `number < 0`: checks if the input is a non-negative integer. Armstrong numbers are typically defined for non-negative integers. If invalid, it prints a message and returns False.
- **Convert to String:** `num_str = str(number)` converts the input number to a string. This makes it easy to:
  - `num_digits = len(num_str)`: Get the number of digits.
  - Iterate through each digit.
- **Calculate Sum of Powers:**
  - `sum_of_powers = 0` initializes a variable to store the sum.
  - The for loop iterates through each character (`digit_char`) in the `num_str`.
  - `digit = int(digit_char)` converts the character back to an integer.
  - `sum_of_powers += digit ** num_digits` calculates the digit raised to the power of the total number of digits and adds it to the running sum.
- **Comparison:** Finally, `return sum_of_powers == number` compares the calculated sum with the original number. If they are equal, the number is an Armstrong number, and the function returns True; otherwise, it returns False.

---

## Comparison Table

Feature	Google Gemini	GitHub Copilot
Logic Style	Uses string conversion	Uses mathematical operations
Code Structure	Function-based	Inline procedural code

Feature	Google Gemini	GitHub Copilot
Readability	Very clear and beginner-friendly	Slightly complex but efficient
Lines of Code	More	Fewer
Ease of Understanding	High	Medium
Suitability for Beginners	Excellent	Good

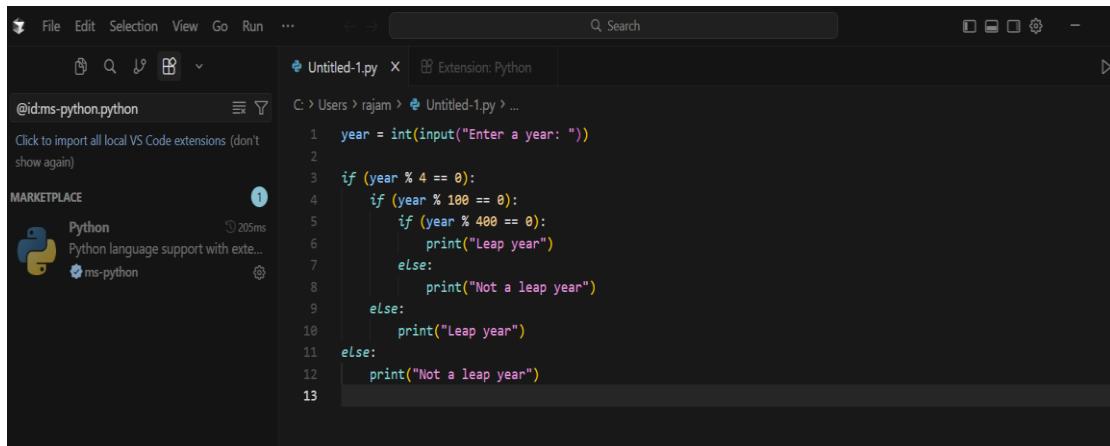
### Task 3: Leap Year Validation Using Cursor AI

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.

#### Prompt 1

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

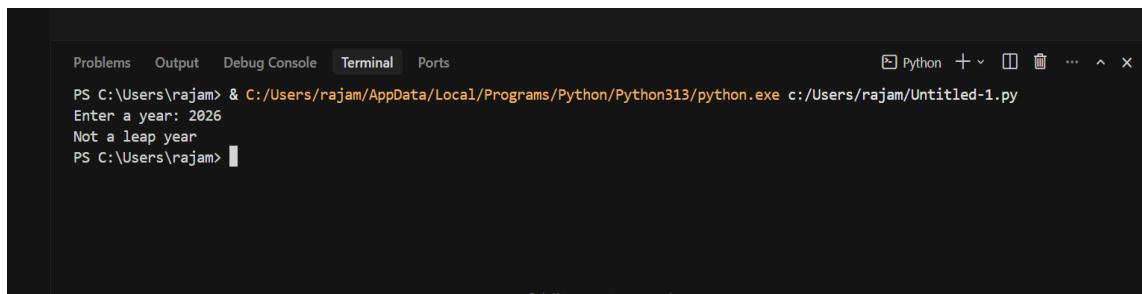


```

@id:ms-python.python
C:\Users\rajam> Untitled-1.py > ...
1  year = int(input("Enter a year: "))
2
3  if (year % 4 == 0):
4      if (year % 100 == 0):
5          if (year % 400 == 0):
6              print("Leap year")
7          else:
8              print("Not a leap year")
9      else:
10         print("Leap year")
11  else:
12      print("Not a leap year")
13

```

#### OUTPUT :



```

PS C:\Users\rajam> & C:/Users/rajam/AppData/Local/Programs/Python/Python313/python.exe c:/Users/rajam/Untitled-1.py
Enter a year: 2026
Not a leap year
PS C:\Users\rajam>

```

#### Prompt 2

Write an optimized and user-friendly Python program to validate leap year with proper comments

The screenshot shows a code editor window with two tabs: 'Untitled-1.py' and 'Untitled-2.py'. The current tab is 'Untitled-2.py'. The code is a Python script for determining if a given year is a leap year. It uses nested if statements to check the divisibility rules for years 4, 100, and 400.

```
1  year = int(input("Enter a year: "))
2  if (year % 4 == 0):
3      if (year % 100 == 0):
4          if (year % 400 == 0):
5              print("Leap year")
6          else:
7              print("Not a leap year")
8      else:
9          print("Leap year")
10 else:
11     print("Not a leap year")
```

## OUTPUT

The screenshot shows a terminal window with several tabs at the top: 'Problems', 'Output', 'Debug Console', 'Terminal' (which is selected), and 'Ports'. The terminal shows the execution of the script 'Untitled-2.py' and its output for two different years.

```
Not a leap year
PS C:\Users\rajam> & C:/Users/rajam/AppData/Local/Programs/Python/Python313/python.exe c:/Users/rajam/Untitled-2.py
Enter a year: 2025
Not a leap year
Not a leap year
PS C:\Users\rajam> & C:/Users/rajam/AppData/Local/Programs/Python/Python313/python.exe c:/Users/rajam/Untitled-2.py
Enter a year: 2024
Leap year
PS C:\Users\rajam>
```

## Comparison Between the Two Versions

Feature	Version 1	Version 2
Prompt Type	Simple	Optimized
Code Structure	Procedural	Function-based

Feature	Version 1	Version 2
Readability	Good	Very Good
Reusability	Low	High
Comments & Documentation	No	Yes
Suitable for Backend Systems	Medium	High

## Conclusion:

Using different prompts in Cursor AI results in improved code quality, better structure, and higher maintainability.

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

Write a Python program that calculates the **sum of odd and even numbers in a tuple**, then refactor it using any AI tool.

### Original code

```
t = (1, 2, 3, 4, 5, 6)

even_sum = 0
odd_sum = 0

for i in t:
    if i % 2 == 0:
        even_sum = even_sum + i
    else:
        odd_sum = odd_sum + i

print("Sum of even numbers:", even_sum)
print("Sum of odd numbers:", odd_sum)
```

### Output

Sum of even numbers: 12

Sum of odd numbers: 9

## Refactored Code

```
t = (1, 2, 3, 4, 5, 6)

even_sum = sum(num for num in t if num % 2 == 0)
odd_sum = sum(num for num in t if num % 2 != 0)

print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")
```

## **Output**

Sum of even numbers: 12

Sum of odd numbers: 9

## **Explanation of Improvements**

Aspect	Original Code	Refactored Code
Code Length	More lines	Fewer lines
Logic Style	Traditional loop	Pythonic (list comprehension)
Readability	Good	Very clear and concise
Performance	Normal	Slightly optimized
Maintainability	Medium	High