

# LAB-02

Name: Zobiya Fatima

Batch No: 14

Roll No: 2303A51879

## Exploring Additional AI Coding Tools beyond Copilot – Gemini (Colab) and Cursor AI

### Task 1: Statistical Summary for Survey Data

#### ❖ Scenario:

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

#### ❖ Task:

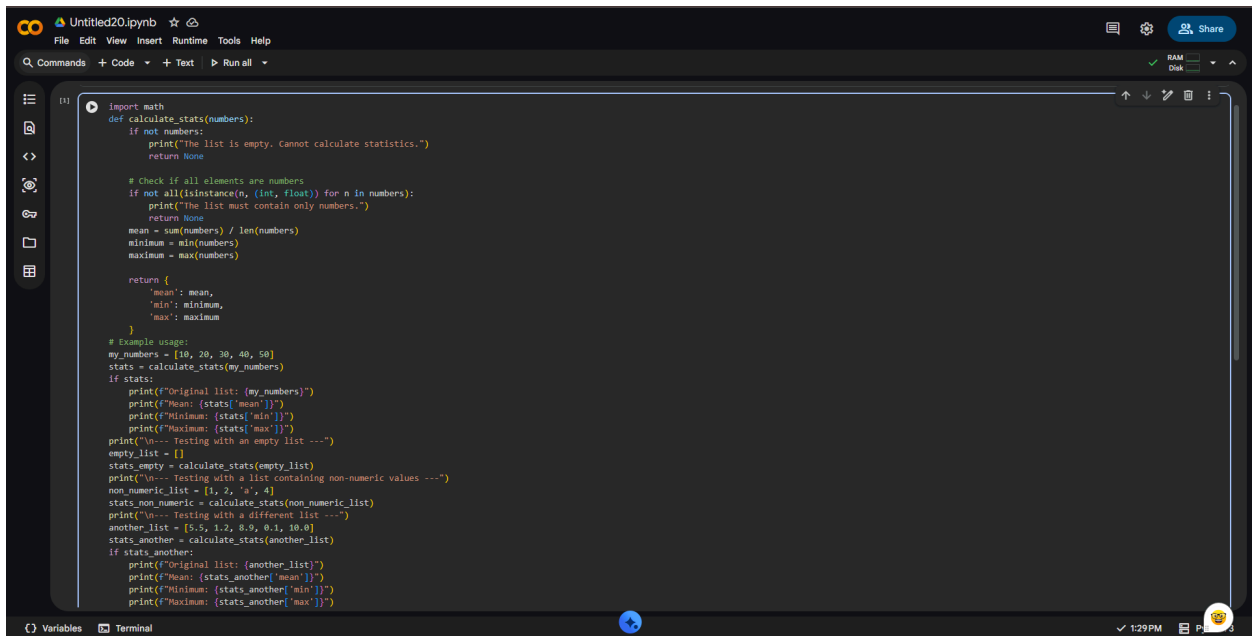
Use Google Gemini in Colab to generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.

#### ❖ Expected Output:

➤ Correct Python function

### Prompt:

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



The screenshot shows a Jupyter Notebook titled 'Untitled20.ipynb'. The code defines a function `calculate_stats` that takes a list of numbers and returns a dictionary with 'mean', 'min', and 'max' values. It includes error handling for empty lists and non-numeric values. The code also demonstrates the function's usage with various test cases.

```
import math

def calculate_stats(numbers):
    if not numbers:
        print("The list is empty. Cannot calculate statistics.")
        return None

    # Check if all elements are numbers
    if not all(isinstance(n, (int, float)) for n in numbers):
        print("The list must contain only numbers.")
        return None

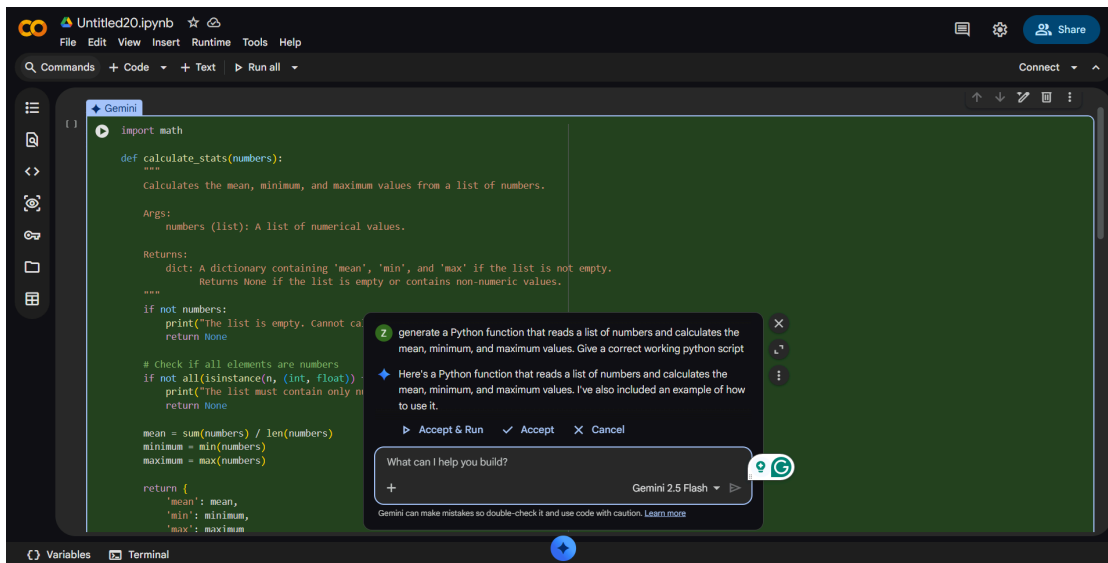
    mean = sum(numbers) / len(numbers)
    minimum = min(numbers)
    maximum = max(numbers)

    return {
        'mean': mean,
        'min': minimum,
        'max': maximum
    }

# Example usage:
my_numbers = [10, 20, 30, 40, 50]
stats = calculate_stats(my_numbers)
if stats:
    print(f"Original list: {my_numbers}")
    print(f"Mean: {stats['mean']}")
    print(f"Minimum: {stats['min']}")
    print(f"Maximum: {stats['max']}")

print("\n--- Testing with an empty list ---")
empty_list = []
stats_empty = calculate_stats(empty_list)
print("\n--- Testing with a list containing non-numeric values ---")
non_numeric_list = [1, 2, 'a', 4]
stats_non_numeric = calculate_stats(non_numeric_list)
print("\n--- Testing with a different list ---")
another_list = [5.5, 1.2, 8.9, 0.1, 10.0]
stats_another = calculate_stats(another_list)
if stats_another:
    print(f"Original list: {another_list}")
    print(f"Mean: {stats_another['mean']}")
    print(f"Minimum: {stats_another['min']}")
    print(f"Maximum: {stats_another['max']}")
```

## ➤ Output shown in Colab



The screenshot shows the same Jupyter Notebook code as above. A Gemini chat overlay is visible, displaying a prompt and a response. The prompt asks for a Python function to calculate mean, minimum, and maximum values. The response provides the function and an example of its usage. The chat interface includes buttons for 'Accept & Run', 'Accept', and 'Cancel'.

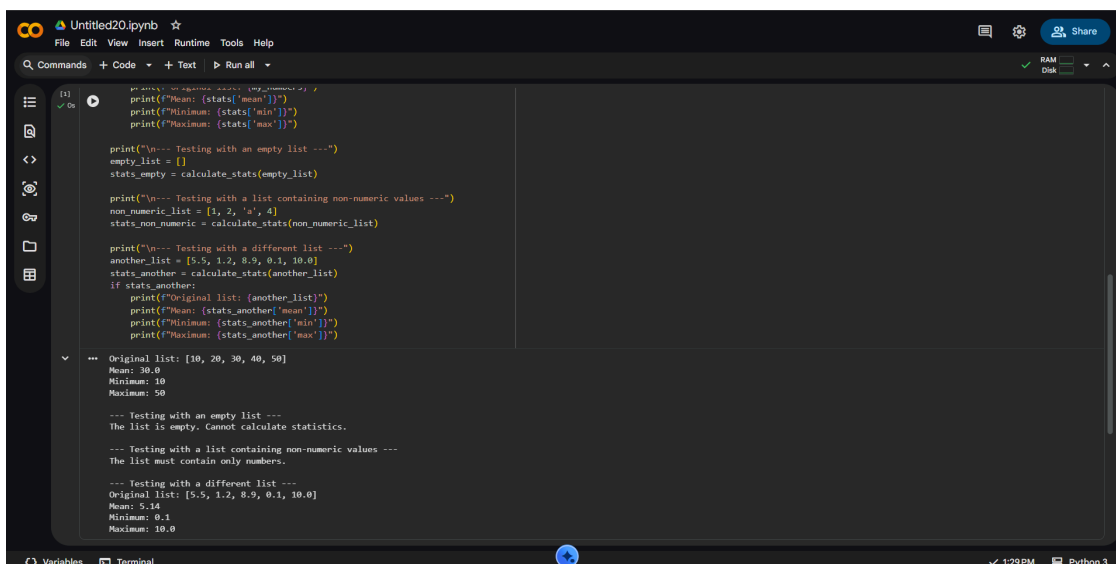
generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values. Give a correct working python script

Here's a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values. I've also included an example of how to use it.

Accept & Run Accept Cancel

What can I help you build?

Gemini 2.5 Flash



The screenshot shows the Jupyter Notebook with the output of the `calculate_stats` function. The output displays the mean, minimum, and maximum values for the original list, and the results of testing with an empty list, a list containing non-numeric values, and a different list.

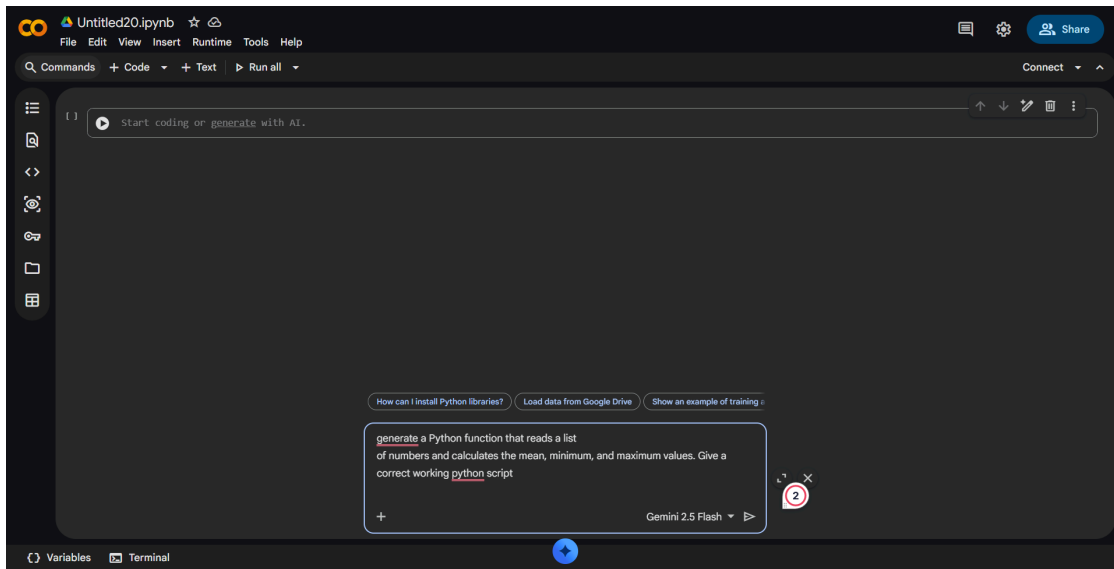
```
Original list: [10, 20, 30, 40, 50]
Mean: 30.0
Minimum: 10
Maximum: 50

--- Testing with an empty list ---
The list is empty. Cannot calculate statistics.

--- Testing with a list containing non-numeric values ---
The list must contain only numbers.

--- Testing with a different list ---
Original list: [5.5, 1.2, 8.9, 0.1, 10.0]
Mean: 5.14
Minimum: 0.1
Maximum: 10.0
```

## ➤ Screenshot of Gemini prompt and result



### Justification:

- The code uses proper input validation to handle empty lists and non-numeric values, preventing runtime errors.
- It provides clear, user-friendly messages instead of failing silently, improving usability.
- The statistical calculations (mean, minimum, maximum) are logically correct, as shown by the test outputs.
- Encapsulating the logic in a function makes the code reusable and easy to test across different datasets.

## Task 2: Armstrong Number – AI Comparison

### ❖ Scenario:

You are evaluating AI tools for numeric validation logic.

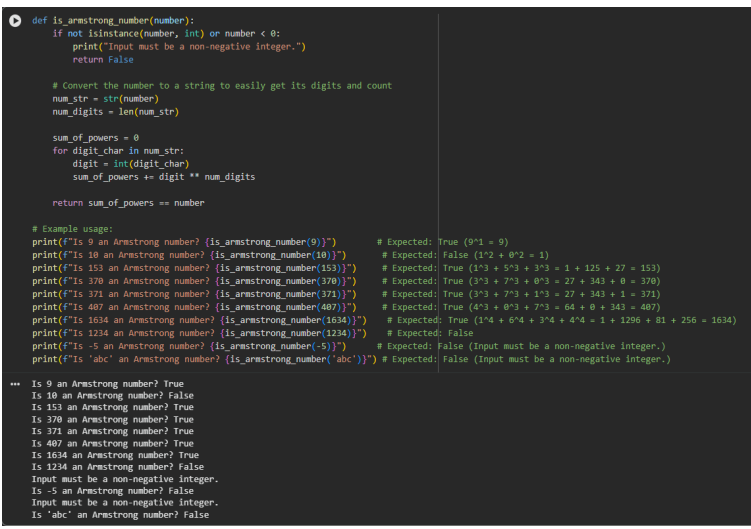
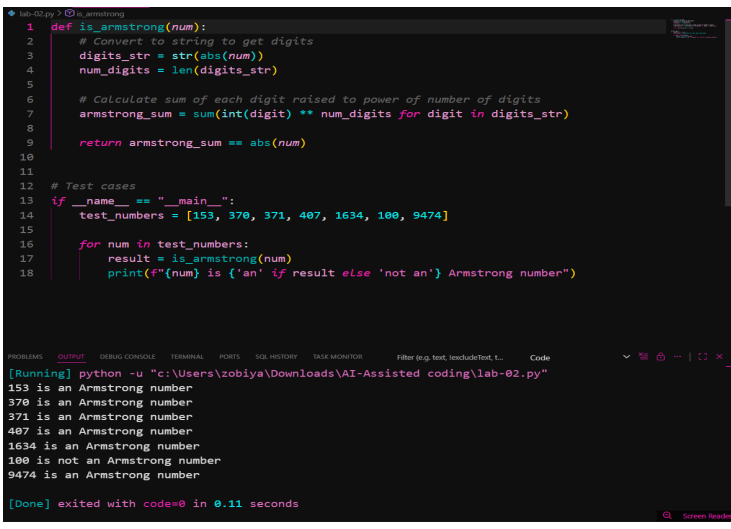
### ❖ Task:

Generate an Armstrong number checker using Gemini and GitHub Copilot.

Compare their outputs, logic style, and clarity.

### ❖ Expected Output:

## ➤ Side-by-side comparison table

Gemini	GitHub Copilot
 <pre>def is_armstrong_number(number):     if not isinstance(number, int) or number &lt; 0:         print("Input must be a non-negative integer.")         return False      # Convert the number to a string to easily get its digits and count     num_str = str(number)     num_digits = len(num_str)      sum_of_powers = 0     for digit_char in num_str:         digit = int(digit_char)         sum_of_powers += digit ** num_digits      return sum_of_powers == number  # Example usage: print(f"Is 9 an Armstrong number? {is_armstrong_number(9)}") # Expected: True (9^1 = 9) print(f"Is 10 an Armstrong number? {is_armstrong_number(10)}") # Expected: False (1^2 + 0^2 = 1) print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}") # Expected: True (1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153) print(f"Is 370 an Armstrong number? {is_armstrong_number(370)}") # Expected: True (3^3 + 7^3 + 0^3 = 27 + 343 + 0 = 370) print(f"Is 371 an Armstrong number? {is_armstrong_number(371)}") # Expected: True (3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371) print(f"Is 407 an Armstrong number? {is_armstrong_number(407)}") # Expected: True (4^3 + 0^3 + 7^3 = 64 + 0 + 343 = 407) print(f"Is 1634 an Armstrong number? {is_armstrong_number(1634)}") # Expected: True (1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634) print(f"Is 1234 an Armstrong number? {is_armstrong_number(1234)}") # Expected: False print(f"Is -5 an Armstrong number? {is_armstrong_number(-5)}") # Expected: False (Input must be a non-negative integer.) print(f"Is 'abc' an Armstrong number? {is_armstrong_number('abc')}") # Expected: False (Input must be a non-negative integer.)  --- Is 9 an Armstrong number? True Is 10 an Armstrong number? False Is 153 an Armstrong number? True Is 370 an Armstrong number? True Is 371 an Armstrong number? True Is 407 an Armstrong number? True Is 1634 an Armstrong number? True Is 1234 an Armstrong number? False Input must be a non-negative integer. Is -5 an Armstrong number? False Input must be a non-negative integer. Is 'abc' an Armstrong number? False</pre>	 <pre>def is_armstrong(num):     # Convert to string to get digits     digits_str = str(abs(num))     num_digits = len(digits_str)      # Calculate sum of each digit raised to power of number of digits     armstrong_sum = sum(int(digit) ** num_digits for digit in digits_str)      return armstrong_sum == abs(num)  # Test cases if __name__ == "__main__":     test_numbers = [153, 370, 371, 407, 1634, 100, 9474]      for num in test_numbers:         result = is_armstrong(num)         print(f"{num} is {'an' if result else 'not an'} Armstrong number")  [Running] python -u "c:\Users\zobiya\Downloads\AI-Assisted coding\lab-02.py" 153 is an Armstrong number 370 is an Armstrong number 371 is an Armstrong number 407 is an Armstrong number 1634 is an Armstrong number 100 is not an Armstrong number 9474 is an Armstrong number  [Done] exited with code=0 in 0.11 seconds</pre>
<b>Prompt:</b> “Generate an Armstrong number checker.”	<b>Prompt:</b> “Generate an Armstrong number checker.”
<b>Output:</b> Produces correct results for valid inputs and provides clear feedback when invalid inputs are given.	<b>Output:</b> Produces correct Armstrong results for valid integers but silently handles negative values by converting them to positive, which may be conceptually misleading.
<b>Logic style:</b> Follows a step-by-step procedural approach, making each stage of the calculation explicit.	<b>Logic style:</b> Uses a compact, Pythonic approach with a generator expression and <code>sum()</code> , prioritizing brevity and efficiency.
<b>Clarity:</b> Uses descriptive variable names and comments, improving readability and understanding.	<b>Clarity:</b> The core logic is condensed into a single line, making it less transparent for beginners.
<b>Error handling:</b> Explicitly checks for non-integer and negative inputs, ensuring logical correctness.	<b>Error handling:</b> Assumes valid input and does not explicitly handle incorrect data types or invalid values.
<b>Use case strength:</b> Better suited as a foundational or educational implementation due to its robustness and clarity.	<b>Use case strength:</b> Well-suited for clean demonstrations or experienced programmers who value concise code.

## ➤ Screenshots of prompts and generated code

### Gemini Code:

```
def is_armstrong_number(number):
    if not isinstance(number, int) or number < 0:
        print("Input must be a non-negative integer.")
        return False

    # Convert the number to a string to easily get its digits and count
    num_str = str(number)
    num_digits = len(num_str)

    sum_of_powers = 0
    for digit_char in num_str:
        digit = int(digit_char)
        sum_of_powers += digit ** num_digits

    return sum_of_powers == number

# Example usage:
print(f"Is 9 an Armstrong number? {is_armstrong_number(9)}") # Expected: True (9^1 = 9)
print(f"Is 10 an Armstrong number? {is_armstrong_number(10)}") # Expected: False (1^2 + 0^2 = 1)
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}") # Expected: True (1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153)
print(f"Is 370 an Armstrong number? {is_armstrong_number(370)}") # Expected: True (3^3 + 7^3 + 0^3 = 27 + 343 + 0 = 370)
print(f"Is 371 an Armstrong number? {is_armstrong_number(371)}") # Expected: True (3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371)
print(f"Is 407 an Armstrong number? {is_armstrong_number(407)}") # Expected: True (4^3 + 0^3 + 7^3 = 64 + 0 + 343 = 407)
print(f"Is 1634 an Armstrong number? {is_armstrong_number(1634)}") # Expected: True (1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634)
print(f"Is 1234 an Armstrong number? {is_armstrong_number(1234)}") # Expected: False
print(f"Is -5 an Armstrong number? {is_armstrong_number(-5)}") # Expected: False (Input must be a non-negative integer.)
print(f"Is 'abc' an Armstrong number? {is_armstrong_number('abc')}") # Expected: False (Input must be a non-negative integer.)

...
Is 9 an Armstrong number? True
Is 10 an Armstrong number? False
Is 153 an Armstrong number? True
Is 370 an Armstrong number? True
Is 371 an Armstrong number? True
Is 407 an Armstrong number? True
Is 1634 an Armstrong number? True
Is 1234 an Armstrong number? False
Input must be a non-negative integer.
Is -5 an Armstrong number? False
Input must be a non-negative integer.
Is 'abc' an Armstrong number? False
```

### GitHub Copilot Code:

```
lab-02.py > is_armstrong
1 def is_armstrong(num):
2     # Convert to string to get digits
3     digits_str = str(abs(num))
4     num_digits = len(digits_str)
5
6     # Calculate sum of each digit raised to power of number of digits
7     armstrong_sum = sum(int(digit) ** num_digits for digit in digits_str)
8
9     return armstrong_sum == abs(num)
10
11
12 # Test cases
13 if __name__ == "__main__":
14     test_numbers = [153, 370, 371, 407, 1634, 100, 9474]
15
16     for num in test_numbers:
17         result = is_armstrong(num)
18         print(f"{num} is {'an' if result else 'not an'} Armstrong number")

[Running] python -u "c:\Users\zobiya\Downloads\AI-Assisted coding\lab-02.py"
153 is an Armstrong number
370 is an Armstrong number
371 is an Armstrong number
407 is an Armstrong number
1634 is an Armstrong number
100 is not an Armstrong number
9474 is an Armstrong number

[Done] exited with code=0 in 0.11 seconds
```

## Justification:

- Focusing on outputs, logic style, and clarity highlights practical aspects that matter in real programming use.
- It shows how Copilot prioritizes conciseness, while Gemini emphasizes correctness and user guidance.
- This structured evaluation makes it easy to identify which code is better suited for learning versus rapid development.
- The comparison separates the codes to fairly assess each AI's design choices without bias or overlap.

## 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:

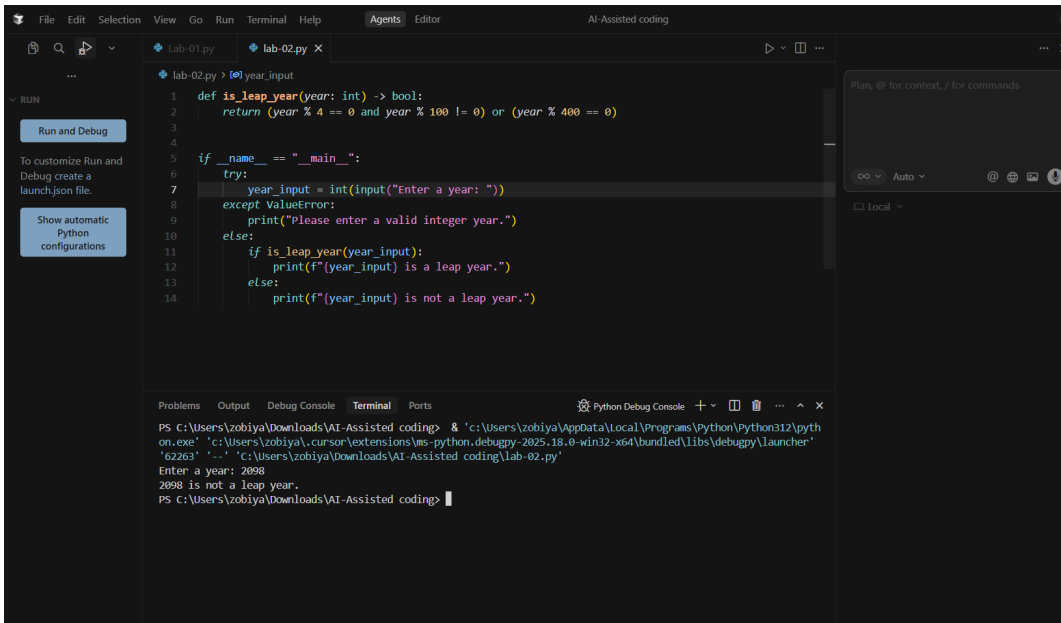
### Prompt:

“generate a Python program that checks whether a given year is a leap year.”

### ➤ Sample inputs/outputs

### ➤ Two versions of code

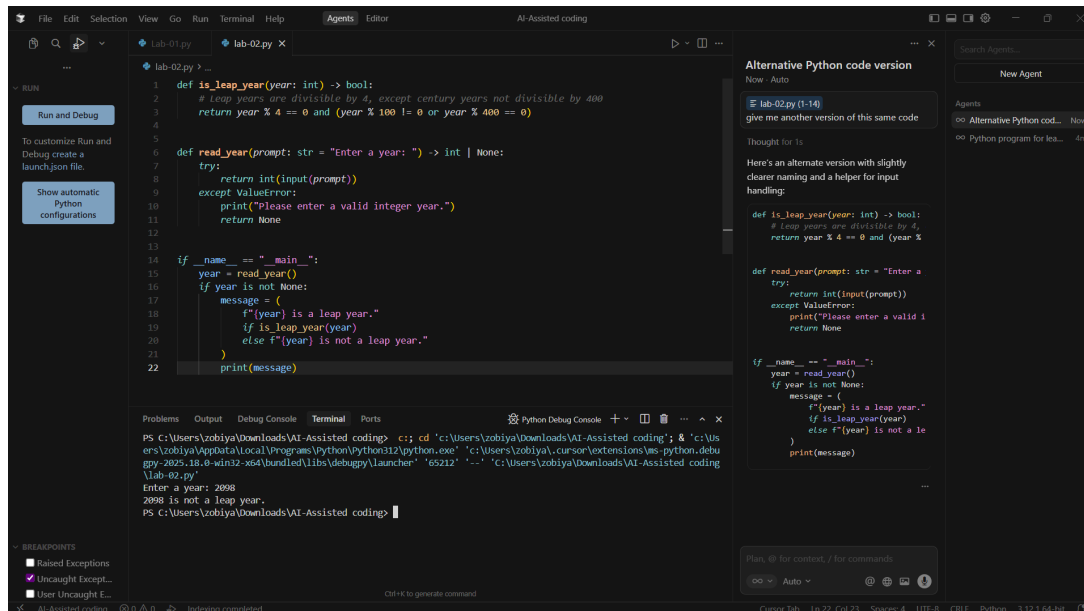
#### ● Version 1:



```
File Edit Selection View Go Run Terminal Help Agents Editor AI-Assisted coding
lab-01.py lab-02.py
lab-02.py > year_input
1 def is_leap_year(year: int) -> bool:
2     return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
3
4
5 if __name__ == "__main__":
6     try:
7         year_input = int(input("Enter a year: "))
8     except ValueError:
9         print("Please enter a valid integer year.")
10    else:
11        if is_leap_year(year_input):
12            print(f"{year_input} is a leap year.")
13        else:
14            print(f"{year_input} is not a leap year.")

Problems Output Debug Console Terminal Ports Python Debug Console
PS C:\Users\zobiya\Downloads\AI-Assisted coding> & 'c:\Users\zobiya\AppData\Local\Programs\Python\Python312\python.exe' 'c:\Users\zobiya\cursor\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundle\libs\debugpy\launcher' '62263' '-.' 'c:\Users\zobiya\Downloads\AI-Assisted coding\lab-02.py'
Enter a year: 2098
2098 is not a leap year.
PS C:\Users\zobiya\Downloads\AI-Assisted coding>
```

## ● Version 2:



## ➤ Brief comparison

Version 2	Version 1
Uses separate functions for leap-year logic and input handling, improving modularity and reuse.	Keeps all logic in a single block, making it short and easy to understand.
Handles invalid input gracefully by returning <b>None</b> , keeping the main logic clean.	Uses a direct <b>try-except</b> for input validation, suitable for quick scripts.
More readable and maintainable for larger or extended programs.	Less modular but faster to write and test.

## Justification:

- The first code is justified by its modular design, which improves readability and reuse.
- The second code is justified by its simplicity, making it quick to write and easy to understand.
- Together, they demonstrate how the same logic can be implemented using scalable versus minimal approaches.

## 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:

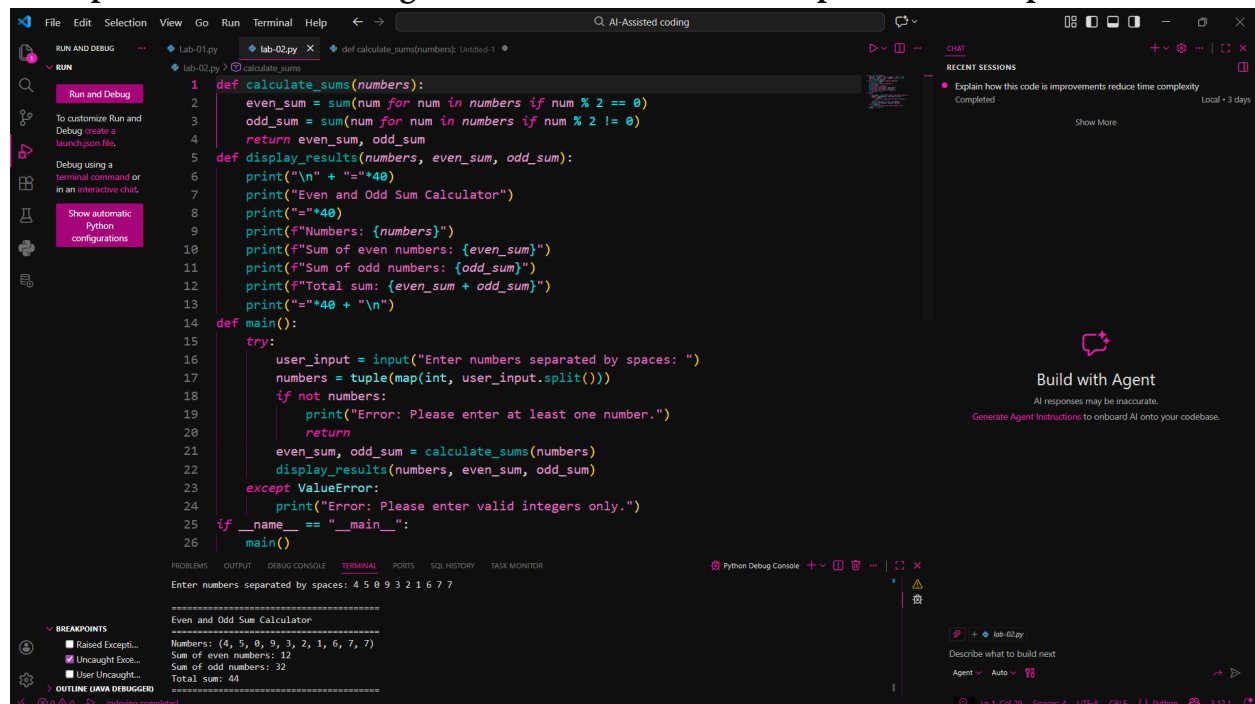
#### ➤ Original code

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9)
even_sum = 0
odd_sum = 0
for num in numbers:
    if num % 2 == 0:
        even_sum = even_sum + num
    else:
        odd_sum = odd_sum + num
print("Tuple:", numbers)
print("Sum of even numbers:", even_sum)
print("Sum of odd numbers:", odd_sum)

... Tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9)
Sum of even numbers: 20
Sum of odd numbers: 25
```

#### ➤ Refactored code

**Prompt:** “Refactor the original code with a better script and user input”



```
1 def calculate_sums(numbers):
2     even_sum = sum(num for num in numbers if num % 2 == 0)
3     odd_sum = sum(num for num in numbers if num % 2 != 0)
4     return even_sum, odd_sum
5
6 def display_results(numbers, even_sum, odd_sum):
7     print("\n" + "="*40)
8     print("Even and Odd Sum Calculator")
9     print("="*40)
10    print(f"Numbers: {numbers}")
11    print(f"Sum of even numbers: {even_sum}")
12    print(f"Sum of odd numbers: {odd_sum}")
13    print(f"Total sum: {even_sum + odd_sum}")
14    print("="*40 + "\n")
15
16 def main():
17     try:
18         user_input = input("Enter numbers separated by spaces: ")
19         numbers = tuple(map(int, user_input.split()))
20         if not numbers:
21             print("Error: Please enter at least one number.")
22             return
23         even_sum, odd_sum = calculate_sums(numbers)
24         display_results(numbers, even_sum, odd_sum)
25     except ValueError:
26         print("Error: Please enter valid integers only.")
27
28 if __name__ == "__main__":
29     main()
```

Enter numbers separated by spaces: 4 5 0 9 3 2 1 6 7 7

Even and Odd Sum Calculator

Numbers: (4, 5, 0, 9, 3, 2, 1, 6, 7, 7)

Sum of even numbers: 32

Sum of odd numbers: 32

Total sum: 64



## ➤ Explanation of improvements

### 1. **User input instead of hardcoded data**

The beginner code uses a fixed tuple, while the improved code allows users to enter numbers dynamically, making it more flexible and practical.

### 2. **Better structure using functions**

The improved code separates logic into functions (`calculate_sums`, `display_results`, `main`), whereas the beginner code puts everything in one block. This improves readability and reuse.

### 3. **Efficient calculation**

The improved version uses Python's `sum()` with conditions to compute even and odd sums concisely, while the beginner version uses a loop and manual accumulation.

### 4. **Error handling and validation**

The improved code handles invalid input (non-integers or empty input) using `try-except`, which the beginner code does not address.

### 5. **Cleaner and more user-friendly output**

The improved code formats the output neatly with headings and totals, making results clearer compared to basic print statements.

## Justification:

The improved code is more flexible because it accepts user input instead of using fixed values.

It is better structured by using functions, which makes the program easier to read and reuse.

Error handling and cleaner output make it more reliable and user-friendly than the beginner version.