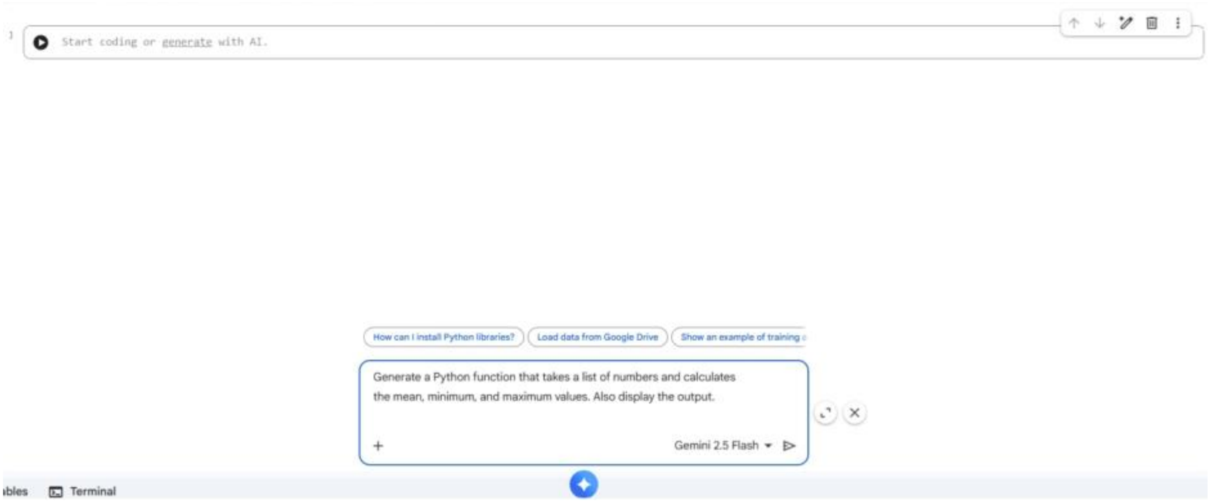
2.1

Name: K Pavani

Batch:27
Hall-Ticket number:2303A51494

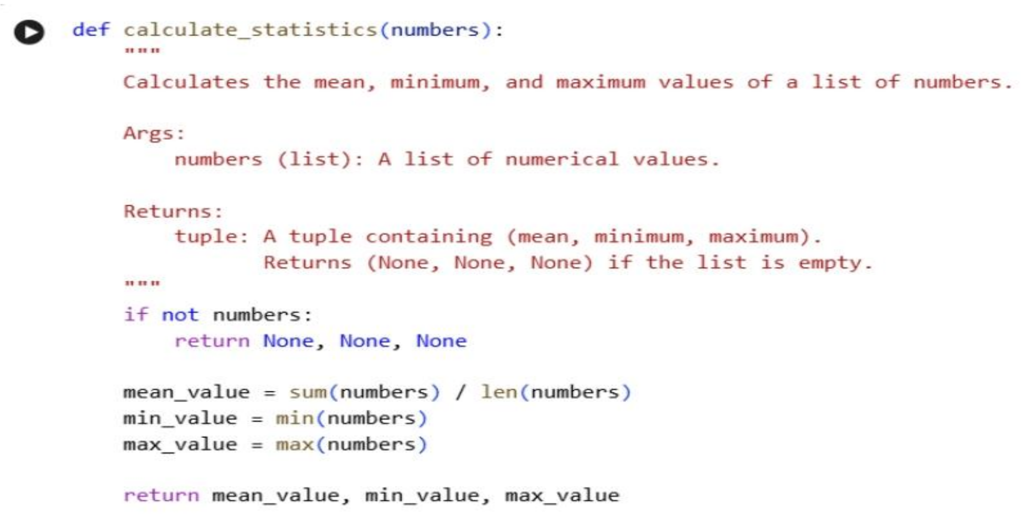
Task1: Statistical Summary for Survey Data

Prompt: Generate a Python function that takes a list of numbers and calculates the mean, minimum, and maximum values. Also display the output

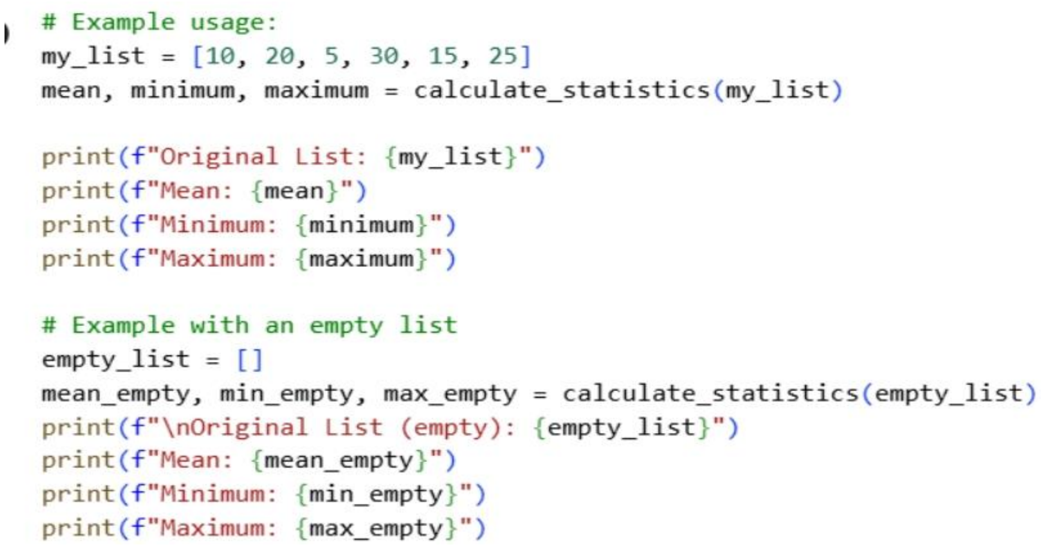

Start coding or generate with AI.
How can I install Python libraries?
Load data from Google Drive
Show an example of training
Gemini 2.5 Flash
ables
Terminal

Generate a Python function that takes a list of numbers and calculates the mean, minimum, and maximum values. Also display the output.

Code:

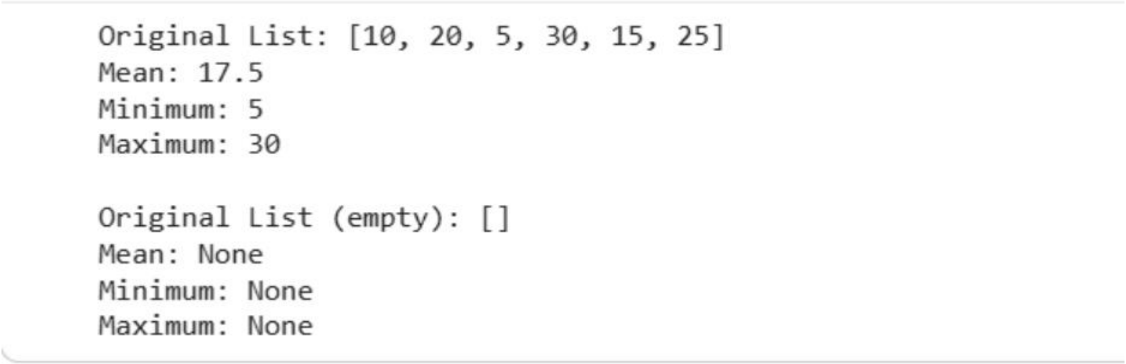


```
def calculate_statistics(numbers):
    """
    Calculates the mean, minimum, and maximum values of a list of numbers.

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

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

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

    return mean_value, min_value, max_value
```



```python
# Example usage:
my_list = [10, 20, 5, 30, 15, 25]
mean, minimum, maximum = calculate_statistics(my_list)

print(f"Original List: {my_list}")
print(f"Mean: {mean}")
print(f"Minimum: {minimum}")
print(f"Maximum: {maximum}")

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

Output:



```
Original List: [10, 20, 5, 30, 15, 25]
Mean: 17.5
Minimum: 5
Maximum: 30

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

Explanation:

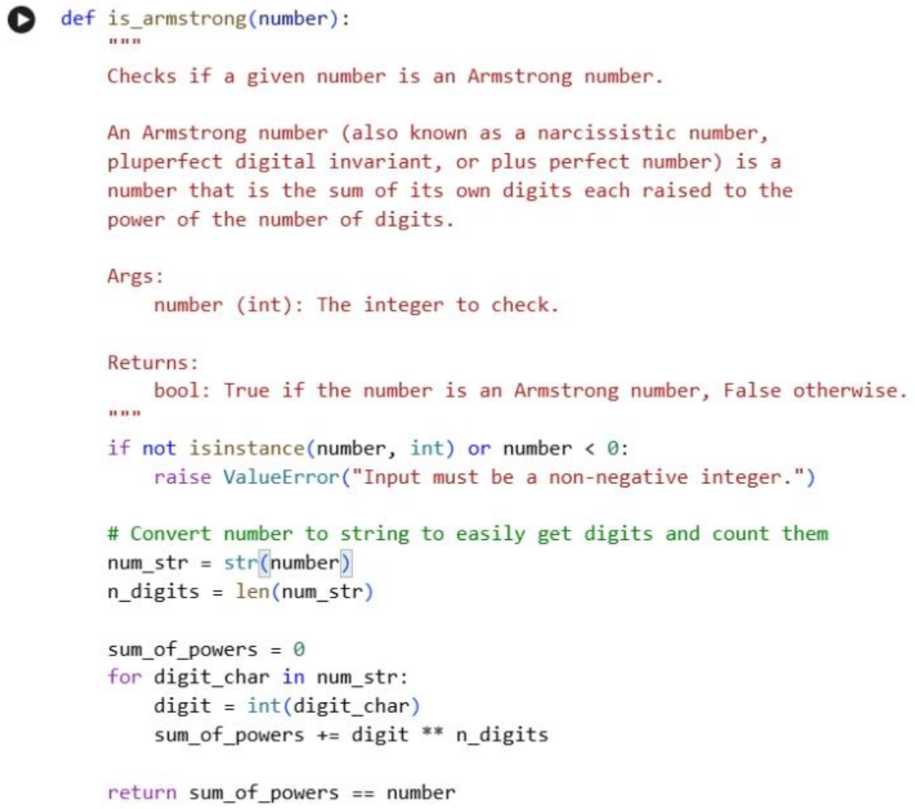
- Creates a reusable function that computes common statistical measures from a given list of values.
- Includes a well-written docstring that describes the function's objective, parameters, and return values.
- Handles the case where the input list is empty to avoid errors such as division by zero.
- Leverages Python's built-in functions to ensure calculations are both efficient and reliable.
- Shows how the function works by running it on a sample dataset for better understanding.
- Confirms proper handling of edge cases by explicitly testing the function with an empty list.

Task2: Armstrong Number – AI Comparison

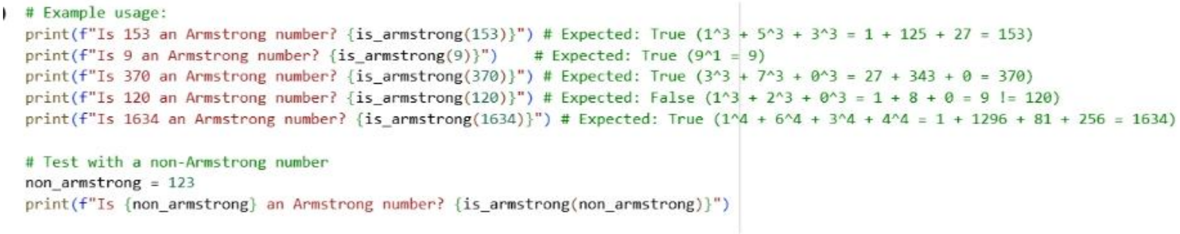
Prompt: Write a Python program to check whether a number is an Armstrong number.

Part A: GEMINI AI(Google Colab)

Code:

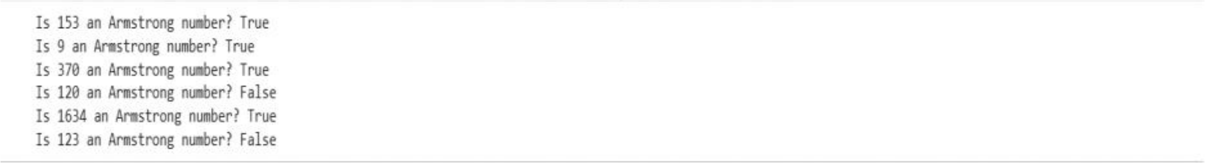


```python
def is_armstrong(number):
    """
    Checks if a given number is an Armstrong number.

    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.

    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:
        raise ValueError("Input must be a non-negative integer.")

    # Convert number to string to easily get digits and count them
    num_str = str(number)
    n_digits = len(num_str)

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

    return sum_of_powers == number
```



```python
# Example usage:
print(f"Is 153 an Armstrong number? {is_armstrong(153)}") # Expected: True (1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153)
print(f"Is 9 an Armstrong number? {is_armstrong(9)}")    # Expected: True (9^1 = 9)
print(f"Is 370 an Armstrong number? {is_armstrong(370)}") # Expected: True (3^3 + 7^3 + 0^3 = 27 + 343 + 0 = 370)
print(f"Is 120 an Armstrong number? {is_armstrong(120)}") # Expected: False (1^3 + 2^3 + 0^3 = 1 + 8 + 0 = 9 != 120)
print(f"Is 1634 an Armstrong number? {is_armstrong(1634)}") # Expected: True (1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634)

# Test with a non-Armstrong number
non_armstrong = 123
print(f"Is {non_armstrong} an Armstrong number? {is_armstrong(non_armstrong)}")
```

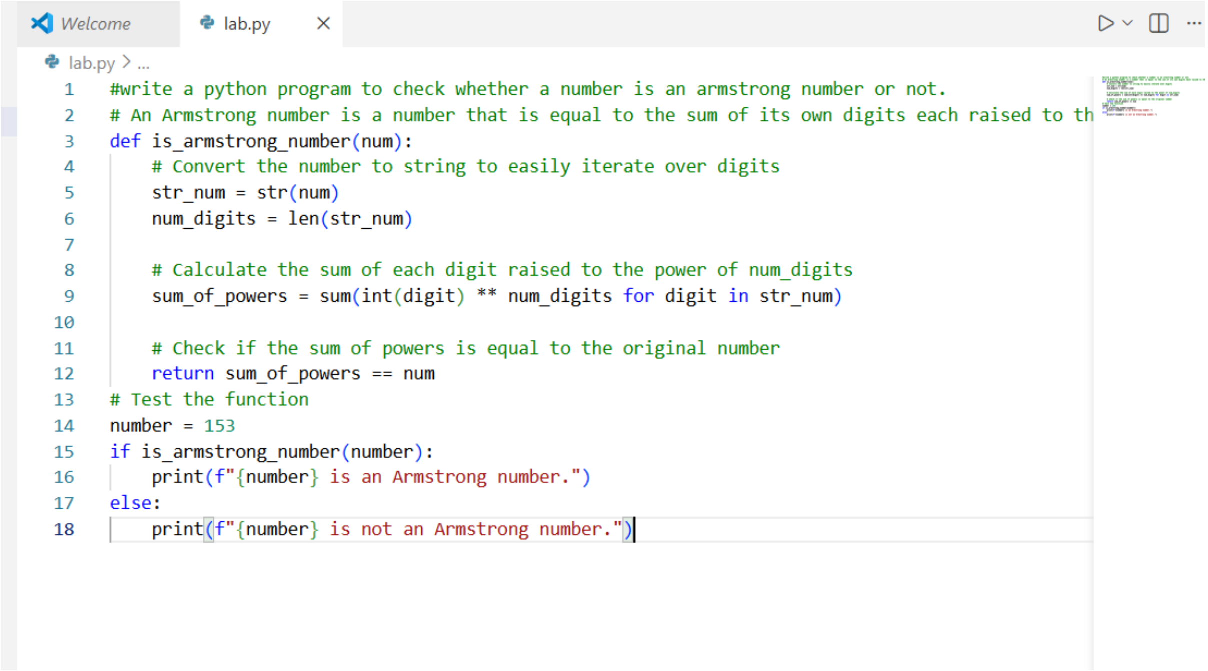
Output:



```
Is 153 an Armstrong number? True
Is 9 an Armstrong number? True
Is 370 an Armstrong number? True
Is 120 an Armstrong number? False
Is 1634 an Armstrong number? True
Is 123 an Armstrong number? False
```

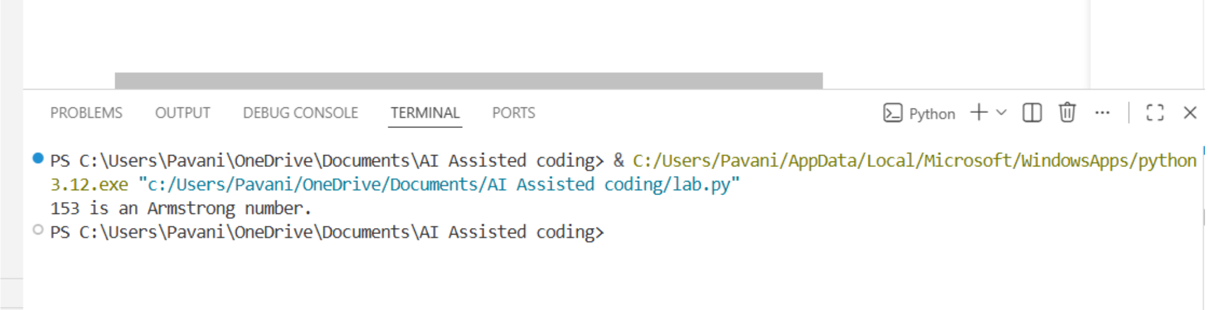
PartB: GitHub Copilot(VS Code)

Code:



```python
#write a python program to check whether a number is an armstrong number or not.
# An Armstrong number is a number that is equal to the sum of its own digits each raised to th
def is_armstrong_number(num):
    # Convert the number to string to easily iterate over digits
    str_num = str(num)
    num_digits = len(str_num)

    # Calculate the sum of each digit raised to the power of num_digits
    sum_of_powers = sum(int(digit) ** num_digits for digit in str_num)

    # Check if the sum of powers is equal to the original number
    return sum_of_powers == num
# Test the function
number = 153
if is_armstrong_number(number):
    print(f"{number} is an Armstrong number.")
else:
    print(f"{number} is not an Armstrong number.")
```

Output:



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python
3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/lab.py"
153 is an Armstrong number.
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

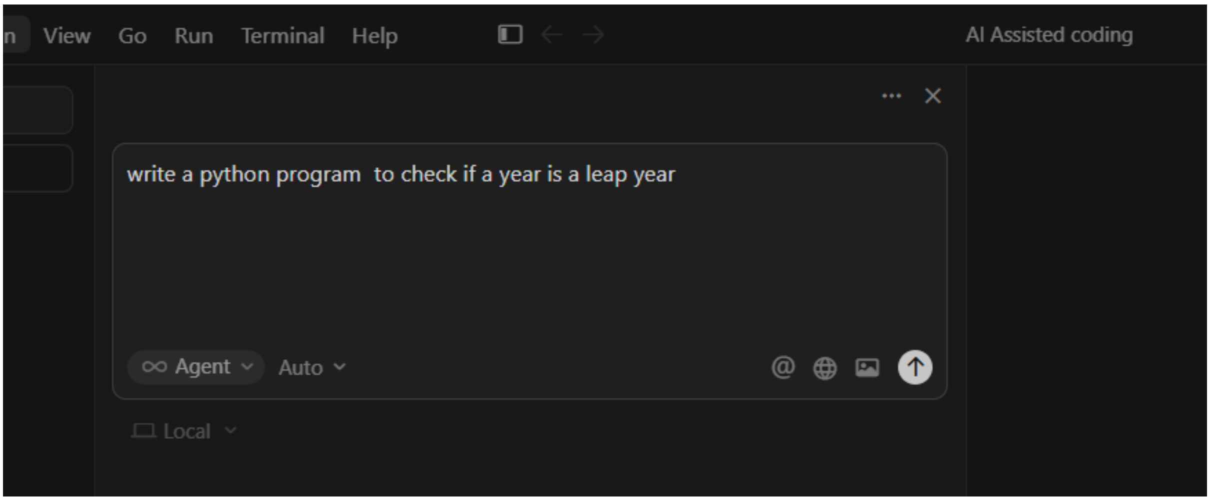
Explanation:

- Gemini AI produces clearer and more structured code, while GitHub Copilot gives correct code but with less organization.
- Both tools use functions, but Gemini includes proper input validation, whereas Copilot adds very few safety checks.
- Gemini’s code is easier to read due to descriptive variable names and good spacing; Copilot’s code is more compact.

- Gemini provides detailed docstrings and inline comments, while Copilot usually includes only minimal comments.
- Gemini follows a direct, step-by-step logic that is easy to understand; Copilot focuses on shorter and optimized expressions.
- Error handling is present in Gemini's code but generally missing in Copilot's output.
- Gemini is more suitable for beginners and learning purposes, while Copilot is better suited for experienced programmers.

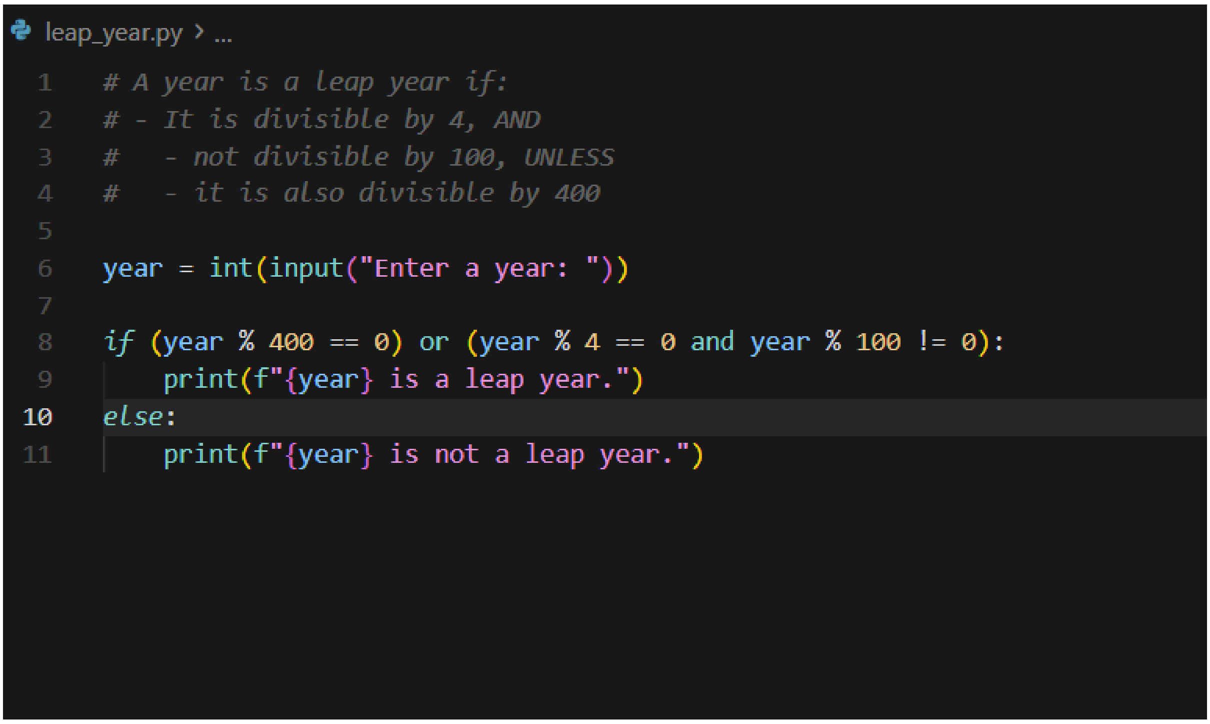
Task3: Leap Year Validation Using Cursor AI

Prompt1: Write a Python program to check if a year is a leap year or not.



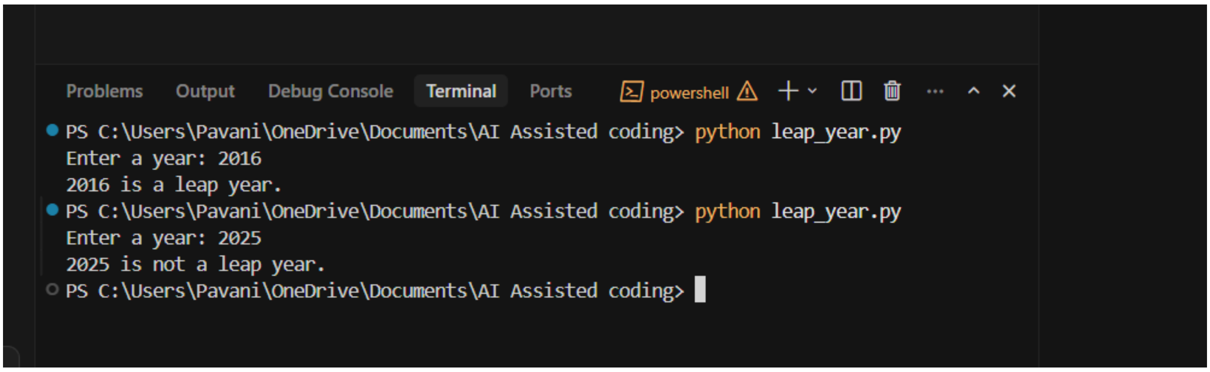
write a python program to check if a year is a leap year

Code:



```python
# A year is a leap year if:
# - It is divisible by 4, AND
#   - not divisible by 100, UNLESS
#   - it is also divisible by 400

year = int(input("Enter a year: "))

if (year % 400 == 0) or (year % 4 == 0 and year % 100 != 0):
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
```

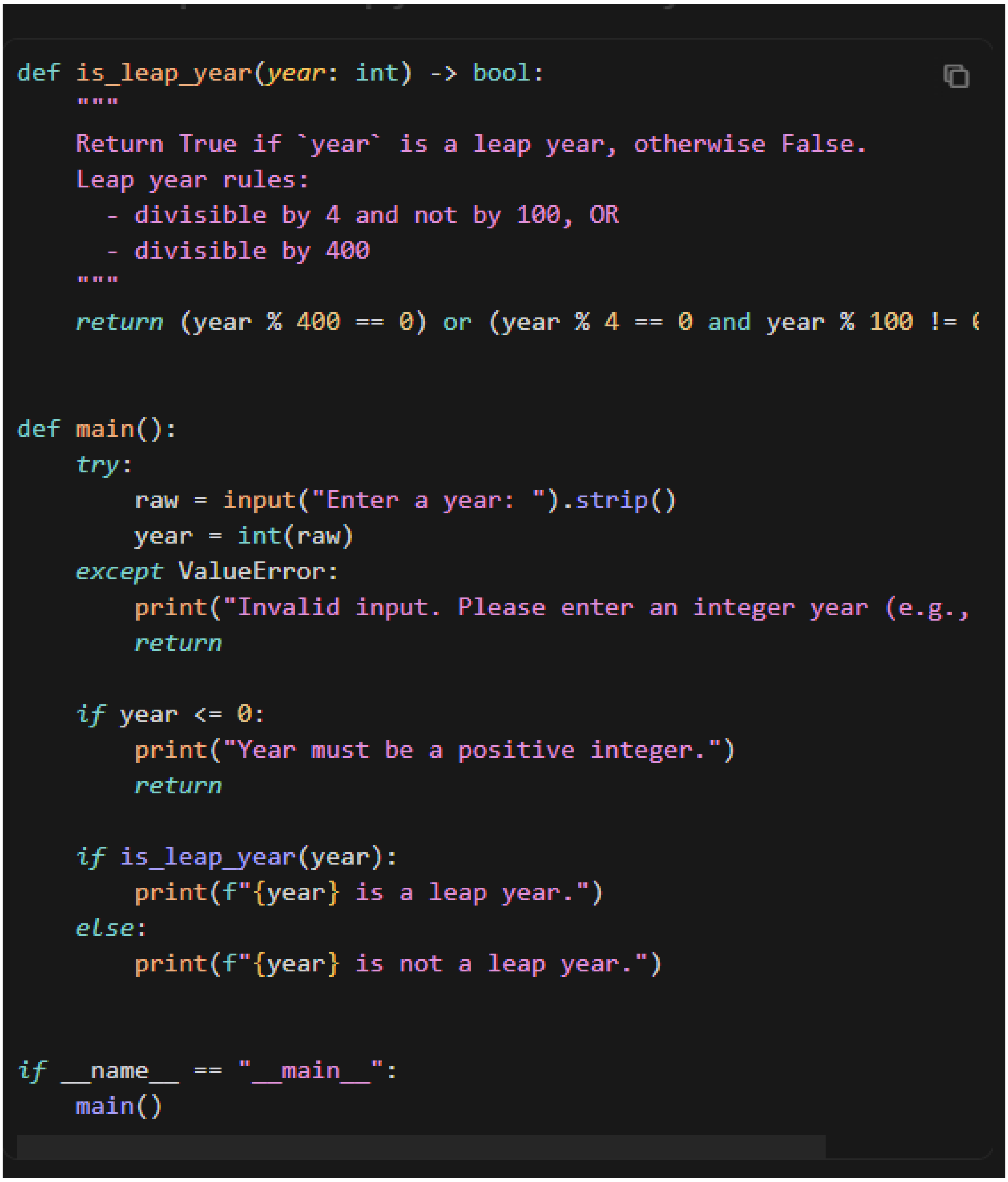
Output :



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> python leap_year.py
Enter a year: 2016
2016 is a leap year.
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> python leap_year.py
Enter a year: 2025
2025 is not a leap year.
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> 
```

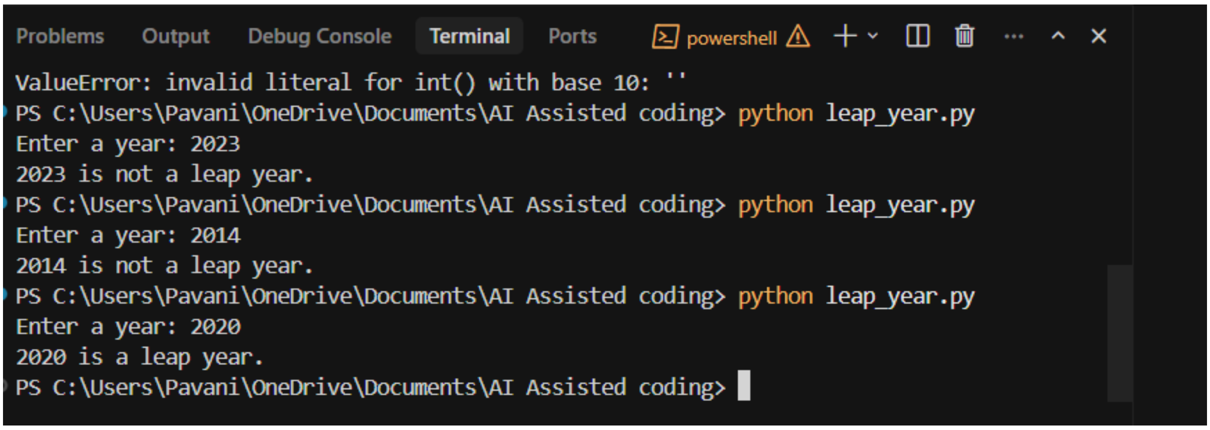
Prompt2: Write a clean and optimized Python function to validate leap year with input and output handling.

Code:



```python
def is_leap_year(year: int) -> bool:
    """
    Return True if `year` is a leap year, otherwise False.
    Leap year rules:
      - divisible by 4 and not by 100, OR
      - divisible by 400
    """
    return (year % 400 == 0) or (year % 4 == 0 and year % 100 != 


def main():
    try:
        raw = input("Enter a year: ").strip()
        year = int(raw)
    except ValueError:
        print("Invalid input. Please enter an integer year (e.g.,
        return

    if year <= 0:
        print("Year must be a positive integer.")
        return

    if is_leap_year(year):
        print(f"{year} is a leap year.")
    else:
        print(f"{year} is not a leap year.")


if __name__ == "__main__":
    main()
```

Output:



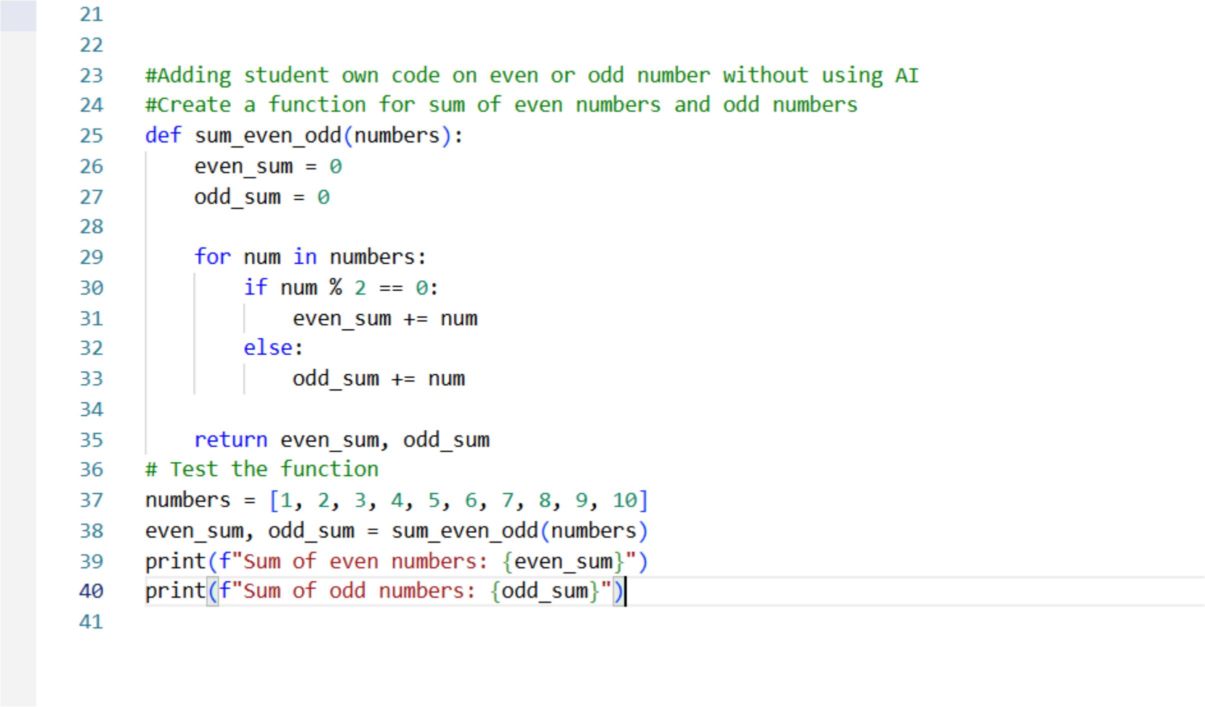
```
ValueError: invalid literal for int() with base 10: ''
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> python leap_year.py
Enter a year: 2023
2023 is not a leap year.
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> python leap_year.py
Enter a year: 2014
2014 is not a leap year.
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> python leap_year.py
Enter a year: 2020
2020 is a leap year.
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> 
```

Explanation:

- The basic version uses nested if–else conditions, which makes the leap year logic easy for beginners to understand.
- The optimized version places the logic inside a function, allowing the code to be reused easily.
- Both versions correctly follow the leap year rules based on divisibility by 4, 100, and 400.
- The optimized code avoids repeated checks and clearly separates the logic from user input and output.
- In summary, the simple code is suitable for learning, while the optimized code is more appropriate for real-world and maintainable applications.

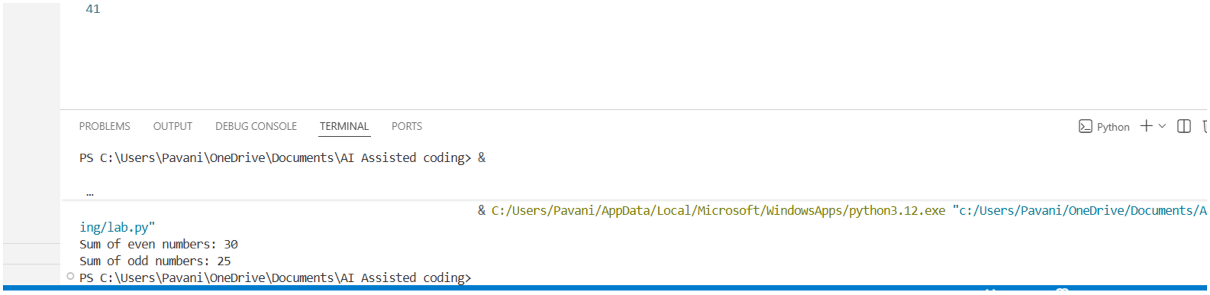
Task4: Student Logic + AI Refactoring (Odd/Even Sum)

Creating Student own code on even Or odd number without using AI



```python


#Adding student own code on even or odd number without using AI
#Create a function for sum of even numbers and odd numbers
def sum_even_odd(numbers):
    even_sum = 0
    odd_sum = 0

    for num in numbers:
        if num % 2 == 0:
            even_sum += num
        else:
            odd_sum += num

    return even_sum, odd_sum
# Test the function
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_sum, odd_sum = sum_even_odd(numbers)
print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")

```

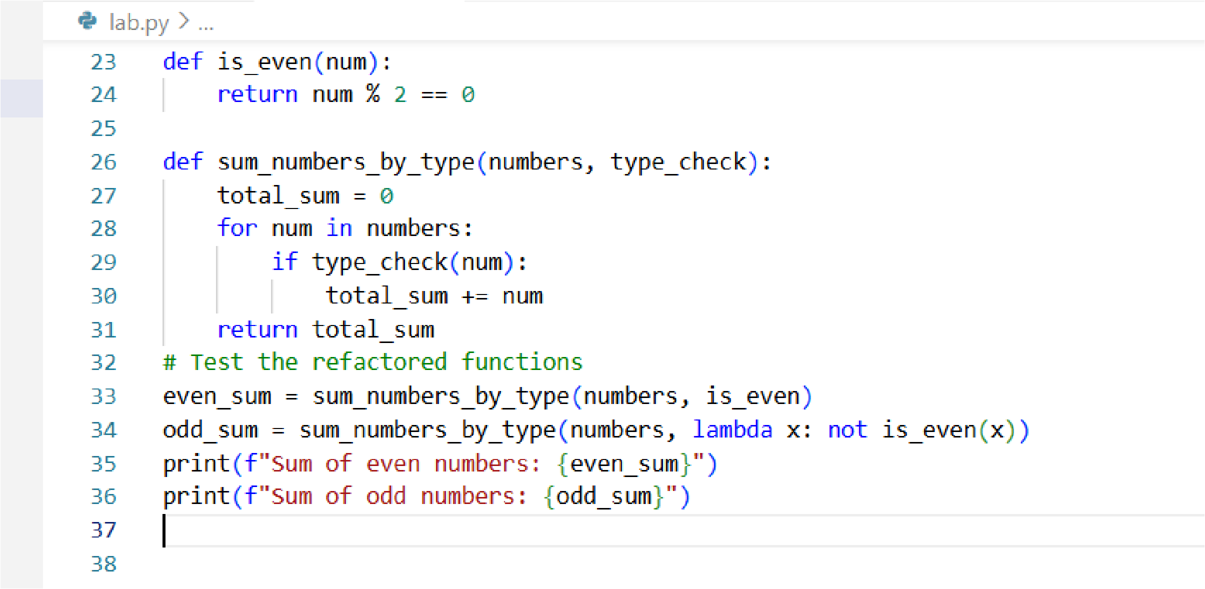
Output:



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> &

 …
                                                  & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/A
ing/lab.py"
Sum of even numbers: 30
Sum of odd numbers: 25
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

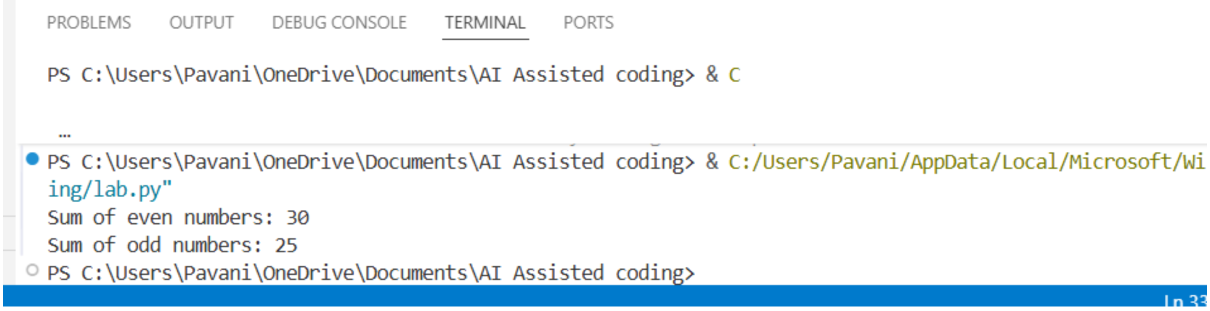
Prompt: Refactor this Python code to make it cleaner, more readable, and reusable.

Code:



```python
def is_even(num):
    return num % 2 == 0

def sum_numbers_by_type(numbers, type_check):
    total_sum = 0
    for num in numbers:
        if type_check(num):
            total_sum += num
    return total_sum
# Test the refactored functions
even_sum = sum_numbers_by_type(numbers, is_even)
odd_sum = sum_numbers_by_type(numbers, lambda x: not is_even(x))
print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")

```

Output:



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C

 ...
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/Wi
ing/lab.py"
Sum of even numbers: 30
Sum of odd numbers: 25
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

Explanation:

- Both programs correctly identify even and odd numbers using the modulus (% 2) operation.
- The first program is written in a straightforward way and prints the output directly, which is easy for beginners to follow.
- The second program is better organized because it includes a docstring and returns results instead of printing them inside the function.
- Returning values makes the optimized program more reusable and suitable for larger applications.
- Overall, the first program helps in understanding basic logic, while the second program follows cleaner and more professional coding practices.