Lab Assignment 2.1

Name: G.Anoop Goud

Hallticket:2303A51085

Batch:02

Task 1: Statistical Summary for Survey Data

Scenario:

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

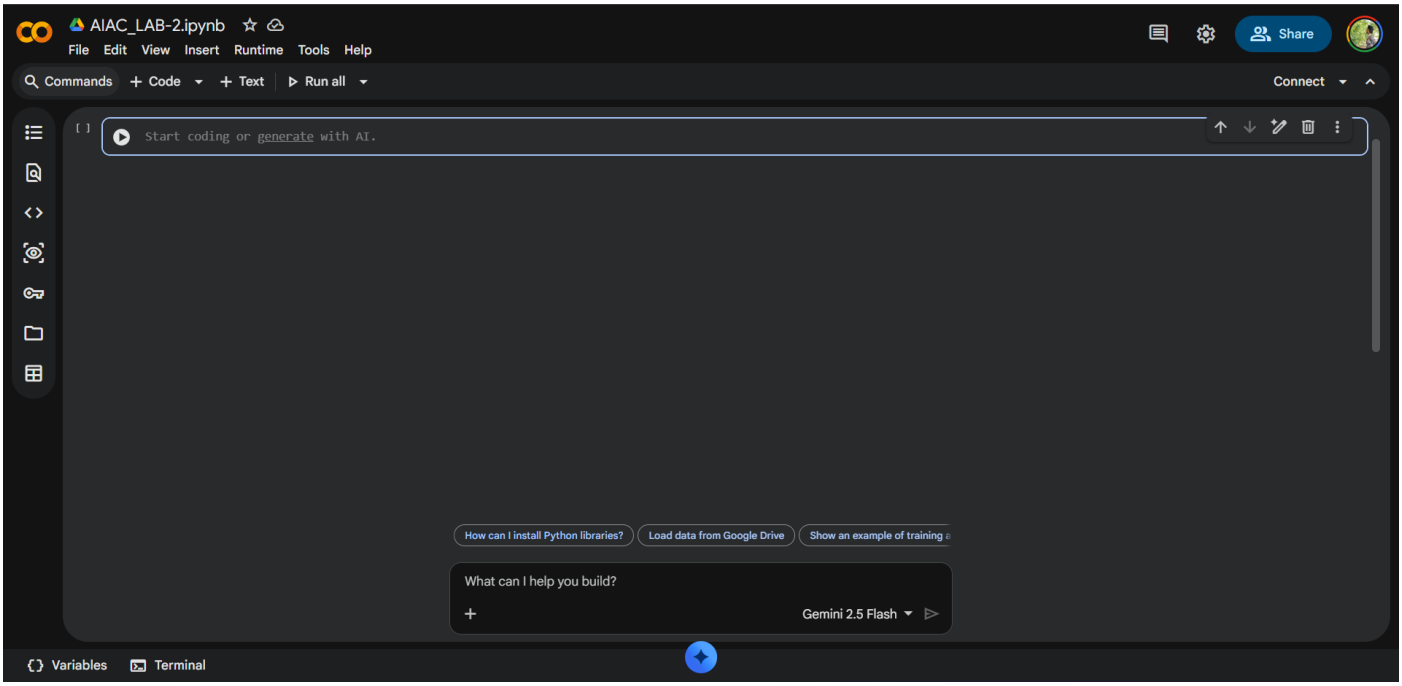
Task:

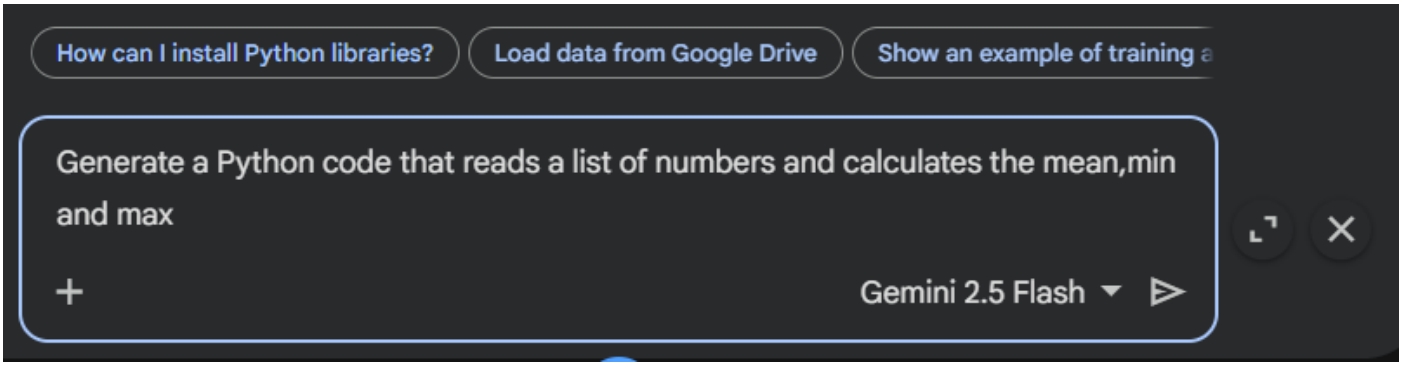
- Use Google Gemini in Colab to generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.

Expected Output:

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

Google Colab Workspace Screenshots:





Code:

```
import statistics

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' values.
              Returns None if the input list is empty.
```

```
    """
    if not numbers:
        return None

    mean_value = statistics.mean(numbers)
    min_value = min(numbers)
    max_value = max(numbers)

    return {
        'mean': mean_value,
        'min': min_value,
        'max': max_value
    }
data = [10, 20, 30, 40, 50, 5, 25]
analysis_results = analyze_numbers(data)

if analysis_results:
    print(f"Original List: {data}")
    print(f"Mean: {analysis_results['mean']}")
    print(f"Minimum: {analysis_results['min']}")
    print(f"Maximum: {analysis_results['max']}")
else:
    print("The list was empty.")

empty_data = []
empty_results = analyze_numbers(empty_data)

if empty_results:
    print(f"Original List: {empty_data}")
    print(f"Mean: {empty_results['mean']}")
    print(f"Minimum: {empty_results['min']}")
```

```
    print(f"Maximum: {empty_results['max']}")
else:
    print(f"Original List: {empty_data}")
    print("The list was empty.")
```

output:

Original List: [10, 20, 30, 40, 50, 5, 25]

Mean: 25.714285714285715

Minimum: 5

Maximum: 50

Original List: []

The list was empty.

Task 2: Armstrong Number – AI Comparison

Scenario:

- You are evaluating AI tools for numeric validation logic.

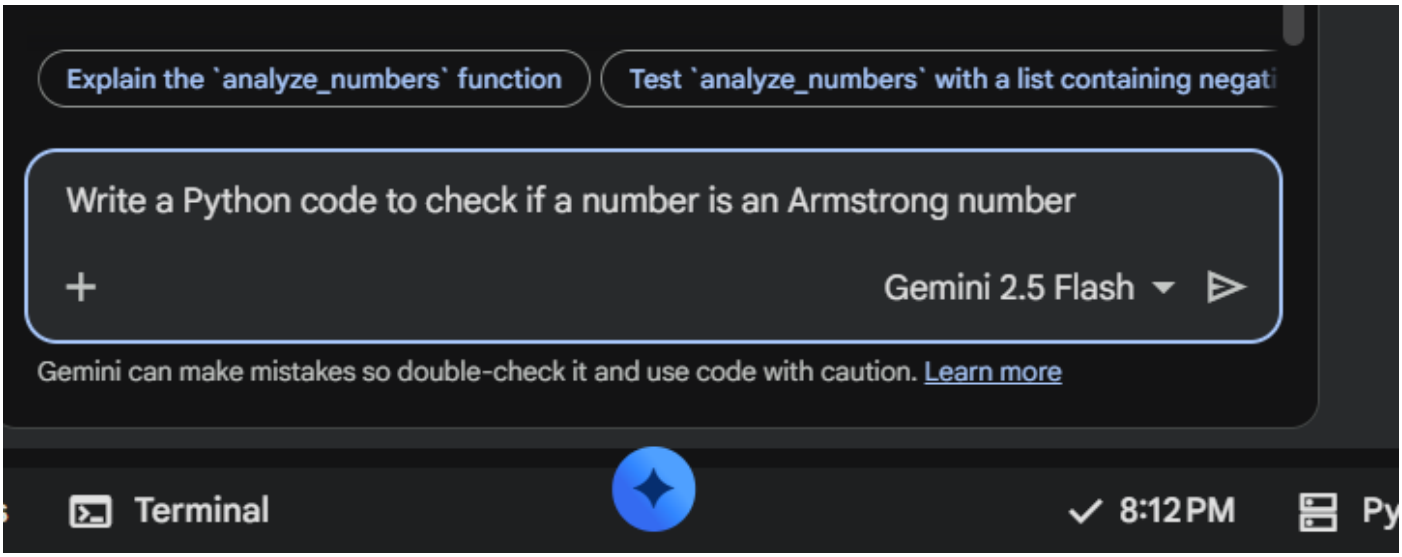
Task:

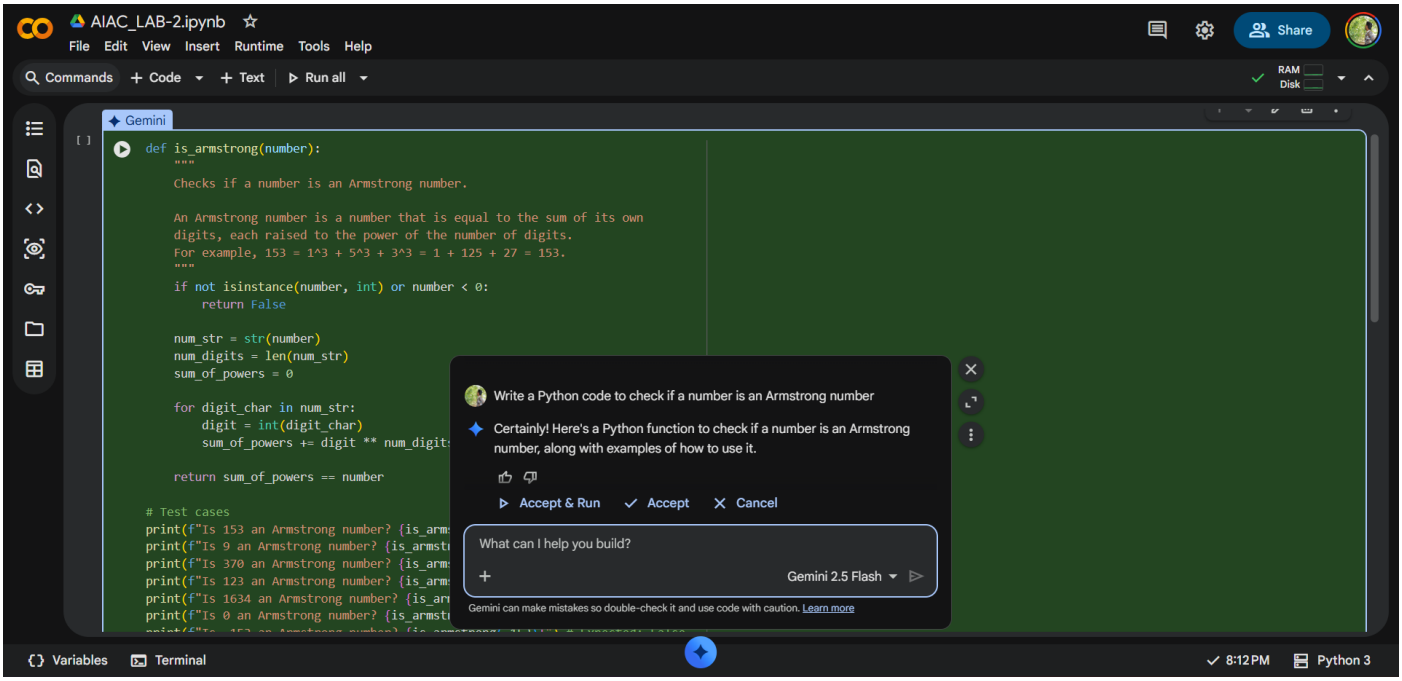
- Generate an Armstrong number checker using Gemini and GitHub Copilot.
- Compare their outputs, logic style, and clarity.

Expected Output:

- Side-by-side comparison table
- Screenshots of prompts and generated code

Google Colab Workspace Screenshots:





Code:

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

    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.
    For example, 153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153.
    """
    if not isinstance(number, int) or number < 0:
        return False

    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


# Test cases
print(f"Is 153 an Armstrong number? {is_armstrong(153)}") # Expected: True
print(f"Is 9 an Armstrong number? {is_armstrong(9)}")   # Expected: True
print(f"Is 370 an Armstrong number? {is_armstrong(370)}") # Expected: True
print(f"Is 123 an Armstrong number? {is_armstrong(123)}") # Expected: False
print(f"Is 1634 an Armstrong number? {is_armstrong(1634)}") # Expected: True
print(f"Is 0 an Armstrong number? {is_armstrong(0)}")   # Expected: True (by some definitions)
print(f"Is -153 an Armstrong number? {is_armstrong(-153)}") # Expected: False
```

output:

Is 153 an Armstrong number? True

Is 9 an Armstrong number? True

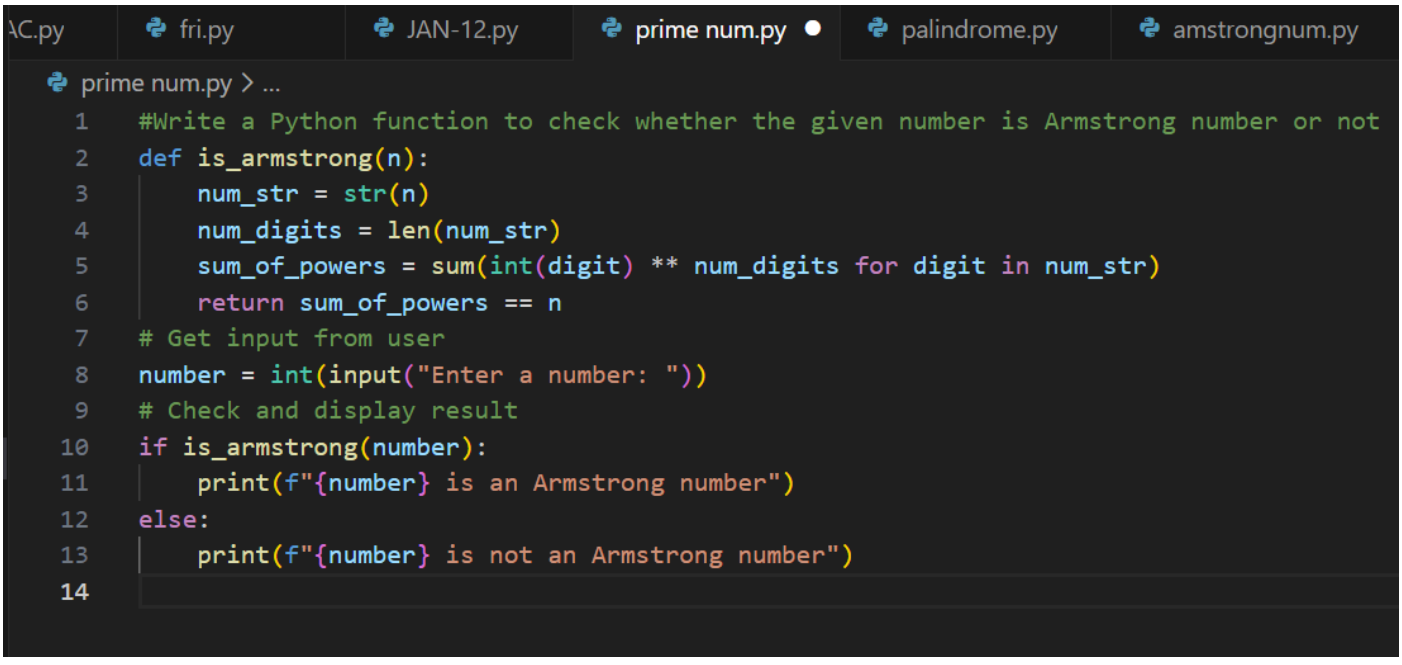
Is 370 an Armstrong number? True

Is 123 an Armstrong number? False

Is 1634 an Armstrong number? True

Is 0 an Armstrong number? True

Is -153 an Armstrong number? FalseVs code :



Code:

```
#Write a Python function to check if a number is an Armstrong number

def is_armstrong(number):

    num_str = str(number)  # Convert the number to string to easily iterate over digits

    num_digits = len(num_str)  # Get the number of digits

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

    return sum_of_powers == number  # Check if the sum of powers is equal to the original number

# Take input from the user

num = int(input("Enter a number to check if it is an Armstrong number: "))

if is_armstrong(num):

    print(f"{num} is an Armstrong number.")

else:

    print(f"{num} is not an Armstrong number.")
```

output:

```
PS C:\Users\anoop\OneDrive\Desktop\AIAC> & C:/Users/anoop/AppData/Local/Programs/P
p/OneDrive/Desktop/AIAC/prime num.py"
Enter a number: 85
85 is not an Armstrong number
PS C:\Users\anoop\OneDrive\Desktop\AIAC>
```

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.

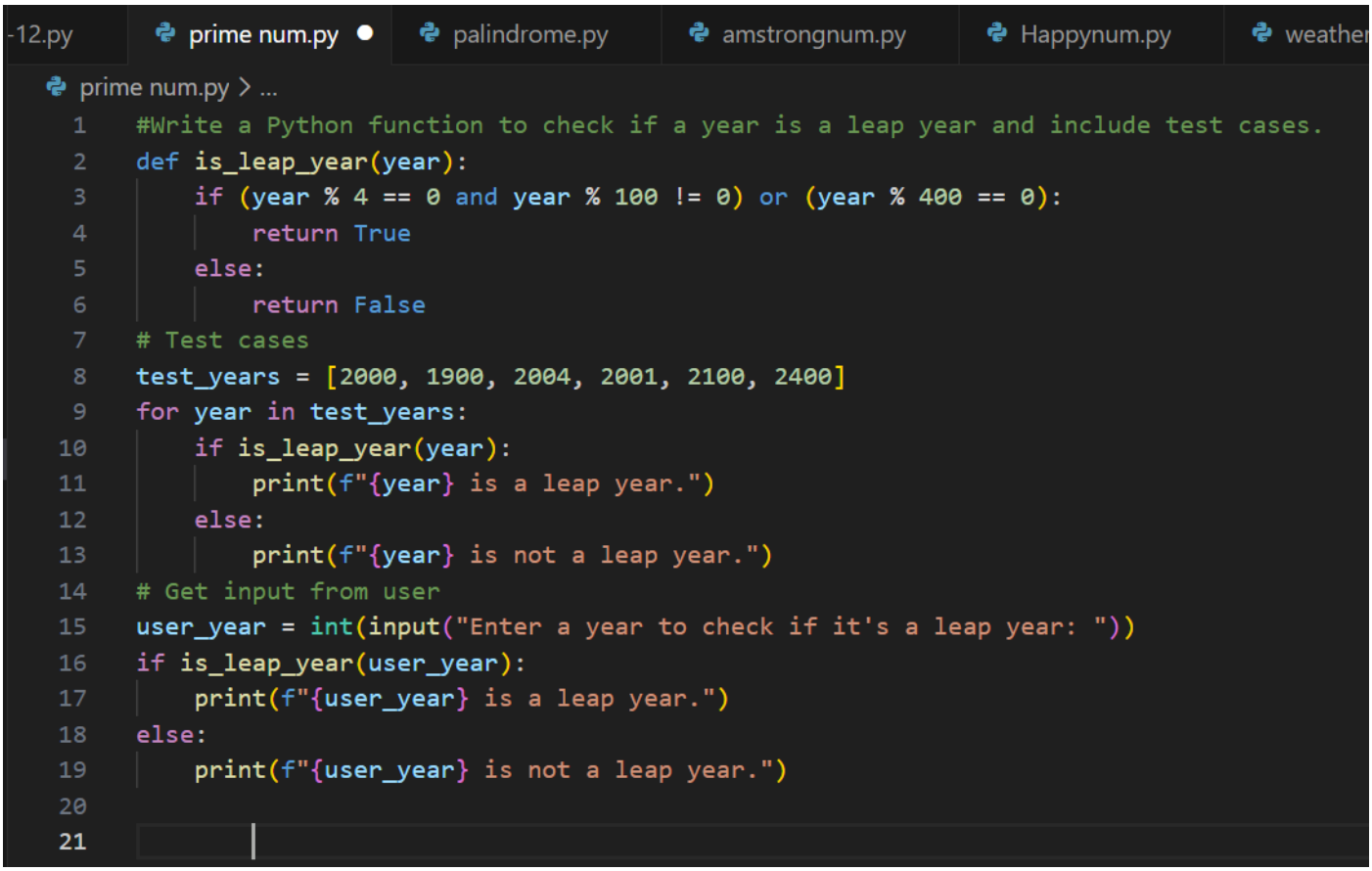
Expected Output:

- Two versions of code
- Sample inputs/outputs
- Brief comparison

Version - 1:

Instruction Given:

Write a Python function to check if a year is a leap year and include test cases.

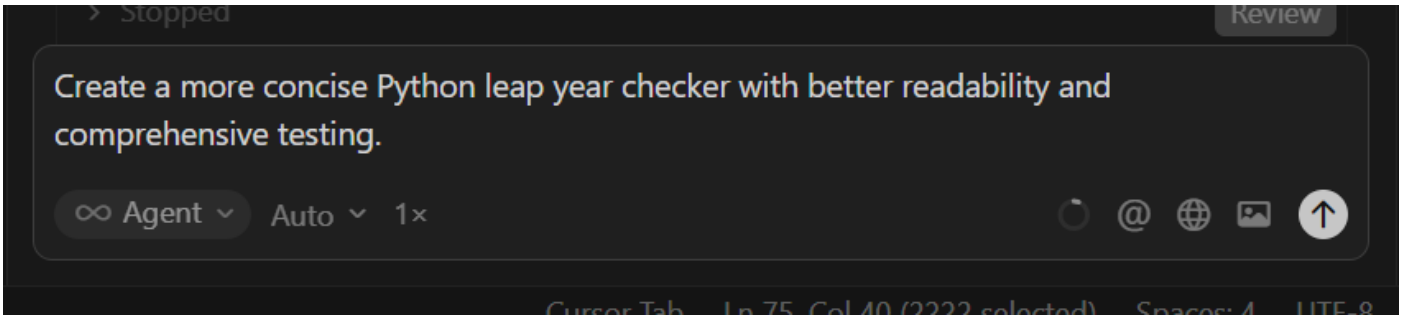


Code:

```
#Write a Python function to check if a year is a leap year and include test cases.
def is_leap_year(year):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    else:
        return False
# Test cases
test_years = [2000, 1900, 2004, 2001, 2100, 2400]
for year in test_years:
    if is_leap_year(year):
        print(f"{year} is a leap year.")
```

```
    else:
        print(f"{year} is not a leap year.")
# Get input from user
user_year = int(input("Enter a year to check if it's a leap year: "))
if is_leap_year(user_year):
    print(f"{user_year} is a leap year.")
else:
    print(f"{user_year} is not a leap year.")
```

Version—02



Code":

```
# Write a Python function to check if a year is a leap year
def is_leap_year(year):
    """
    Checks if a given year is a leap year.

    A year is a leap year if:
    - It is divisible by 4,
    - EXCEPT if it is divisible by 100,
```

```
    - UNLESS it is also divisible by 400.
    """
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    else:
        return False

# Test cases
print(f"Is 2000 a leap year? {is_leap_year(2000)}") # Expected: True (divisible by 400)
print(f"Is 1900 a leap year? {is_leap_year(1900)}") # Expected: False (divisible by 100 but not 400)
print(f"Is 2024 a leap year? {is_leap_year(2024)}") # Expected: True (divisible by 4 and not by 100)
print(f"Is 2023 a leap year? {is_leap_year(2023)}") # Expected: False (not divisible by 4)
print(f"Is 1600 a leap year? {is_leap_year(1600)}") # Expected: True
```

output:

```
Is 2000 a leap year? True
Is 1900 a leap year? False
Is 2024 a leap year? True
Is 2023 a leap year? False
Is 1600 a leap year? True
```

Brief comparison:

Aspect	**Version 1 (Basic)**	**Version 2 (Concise)**
Logic	Nested if-else	Single return with logical operators
Lines of code	More (≈12)	Fewer (≈5)
Readability	Step-by-step	Compact
Beginner friendly	Yes	No
Style	Traditional	Pythonic
Conditions	Separate checks	Combined logic
Testing	Basic tests	Comprehensive tests

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

Original Code written by me:

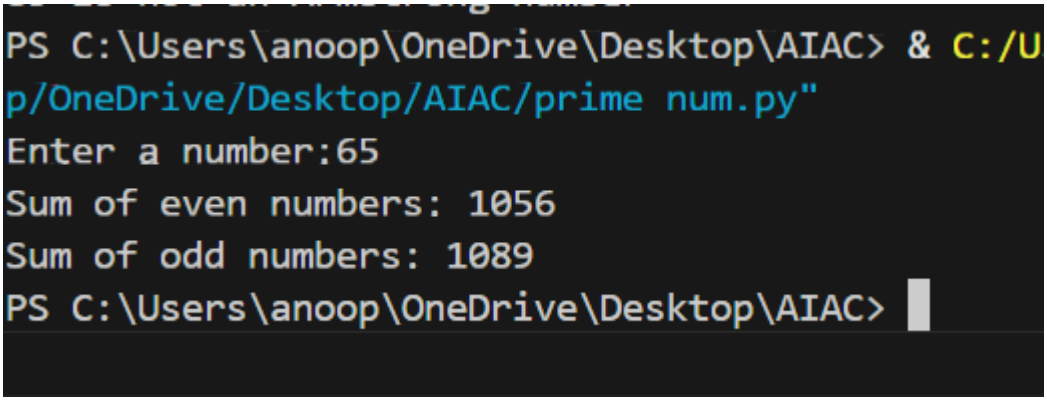
```python
def even_odd_sum_tuple(num):
    even_sum = 0
    odd_sum = 0
    for i in range(1, num+1):
        if i%2 == 0:
            even_sum += i
        else:
            odd_sum += i
    return even_sum, odd_sum
user_input = int(input("Enter a number: "))
even, odd = even_odd_sum_tuple(user_input)
print(f"Sum of even numbers: {even}")
print(f"Sum of odd numbers: {odd}")
```

Code:

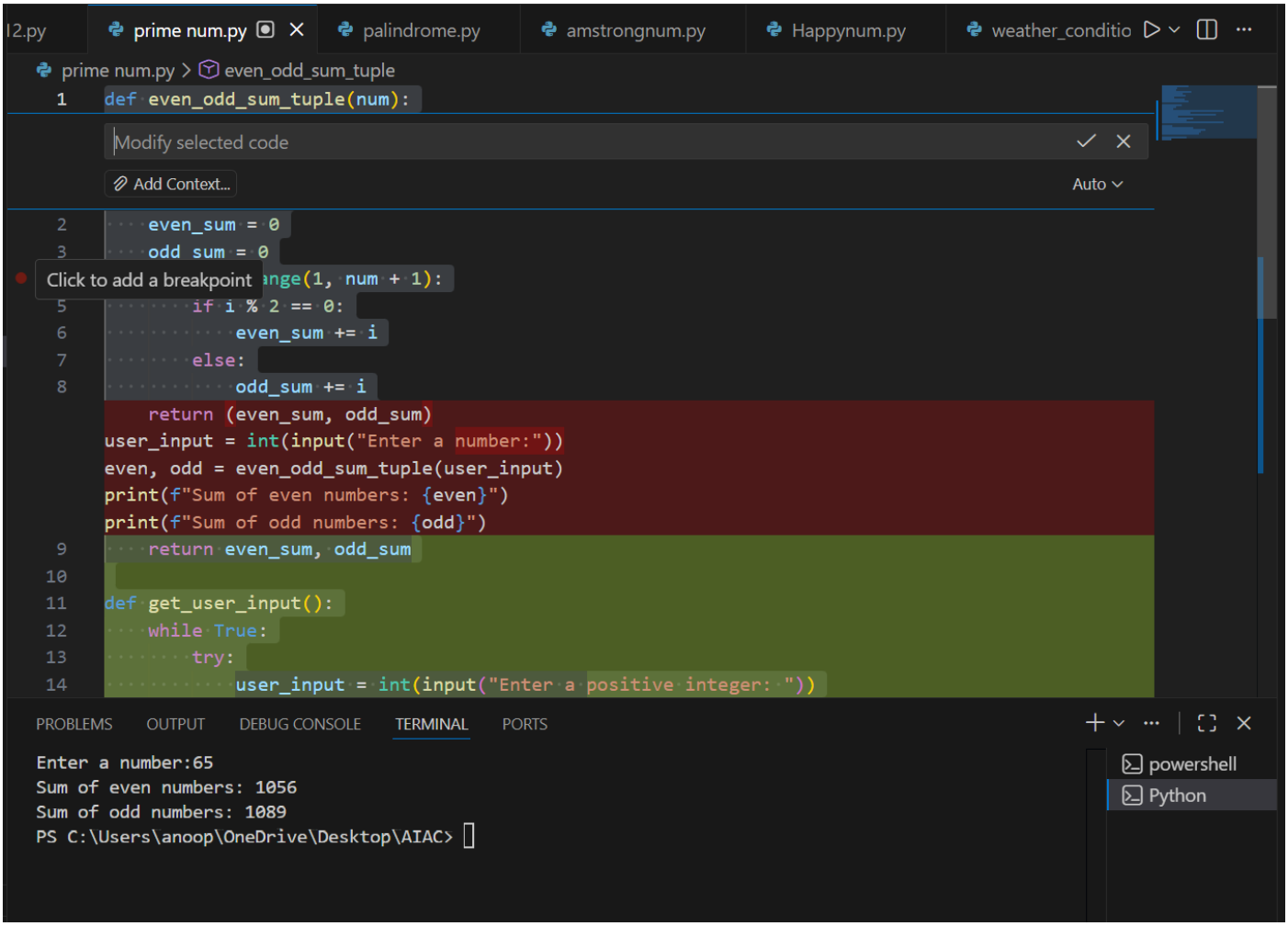
```
def even_odd_sum_tuple(num):
  even_sum = 0
  odd_sum = 0
  for i in range(1, num+1):
    if i%2 == 0:
      even_sum += i
```

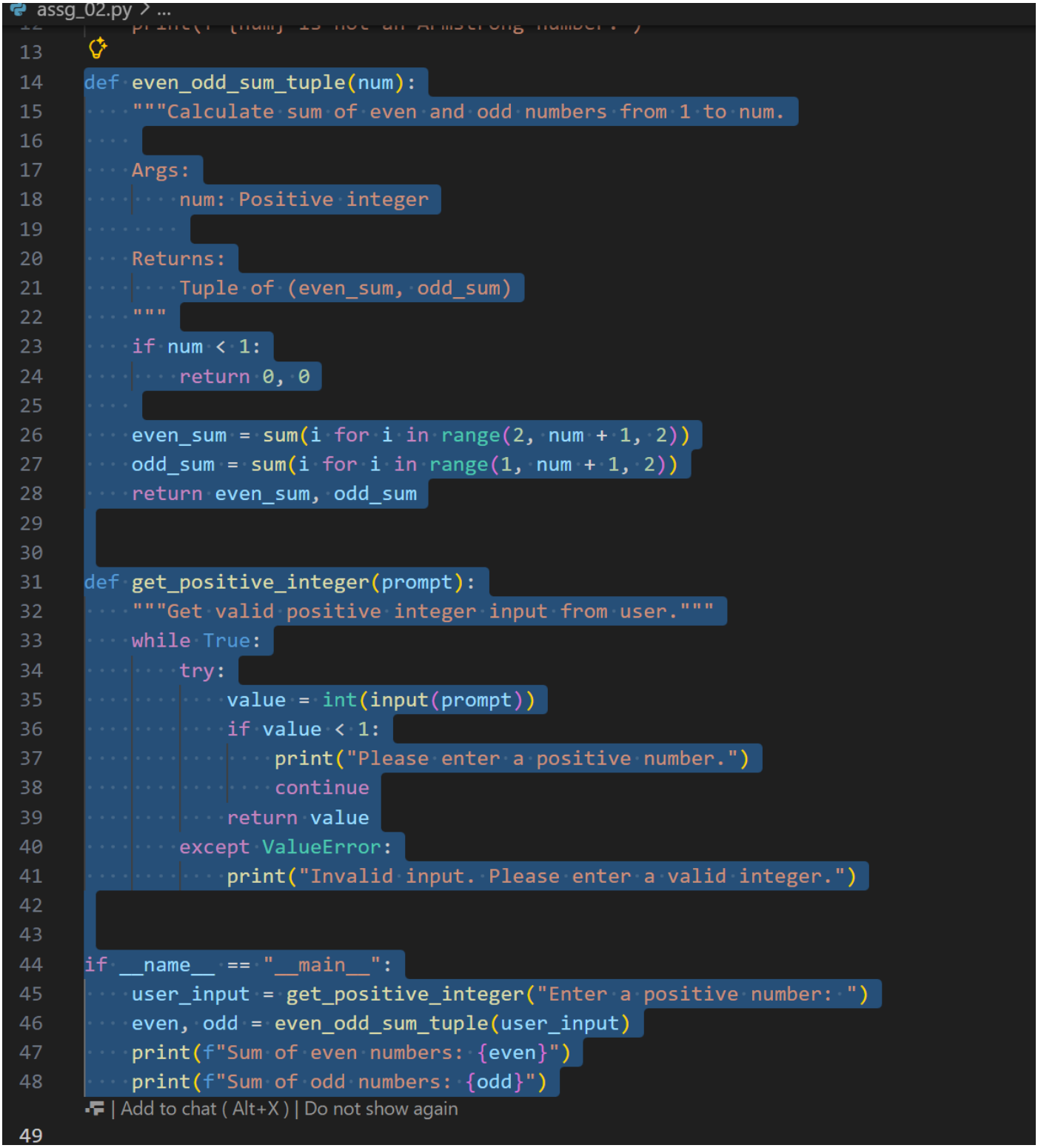
```
        else:
            odd_sum += i
    return even_sum, odd_sum
user_input = int(input("Enter a number: "))
even, odd = even_odd_sum_tuple(user_input)
print(f"Sum of even numbers: {even}")
print(f"Sum of odd numbers: {odd}")
```

Output:



Refactoring the code :


I2.py
prime num.py
palindrome.py
amstrongnum.py
Happynum.py
weather_conditio
prime num.py > even_odd_sum_tuple
def even_odd_sum_tuple(num):
Modify selected code
Add Context...
Auto
even_sum = 0
odd_sum = 0
Click to add a breakpoint
if i % 2 == 0:
even_sum += i
else:
odd_sum += i
return (even_sum, odd_sum)
user_input = int(input("Enter a number:"))
even, odd = even_odd_sum_tuple(user_input)
print(f"Sum of even numbers: {even}")
print(f"Sum of odd numbers: {odd}")
return even_sum, odd_sum
def get_user_input():
while True:
try:
user_input = int(input("Enter a positive integer: "))
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Enter a number:65
Sum of even numbers: 1056
Sum of odd numbers: 1089
PS C:\Users\anoop\OneDrive\Desktop\AIAC>
powershell
Python



Code

```
def even_odd_sum_tuple(num):
    """Calculate sum of even and odd numbers from 1 to num.

    Args:
        num: Positive integer
```

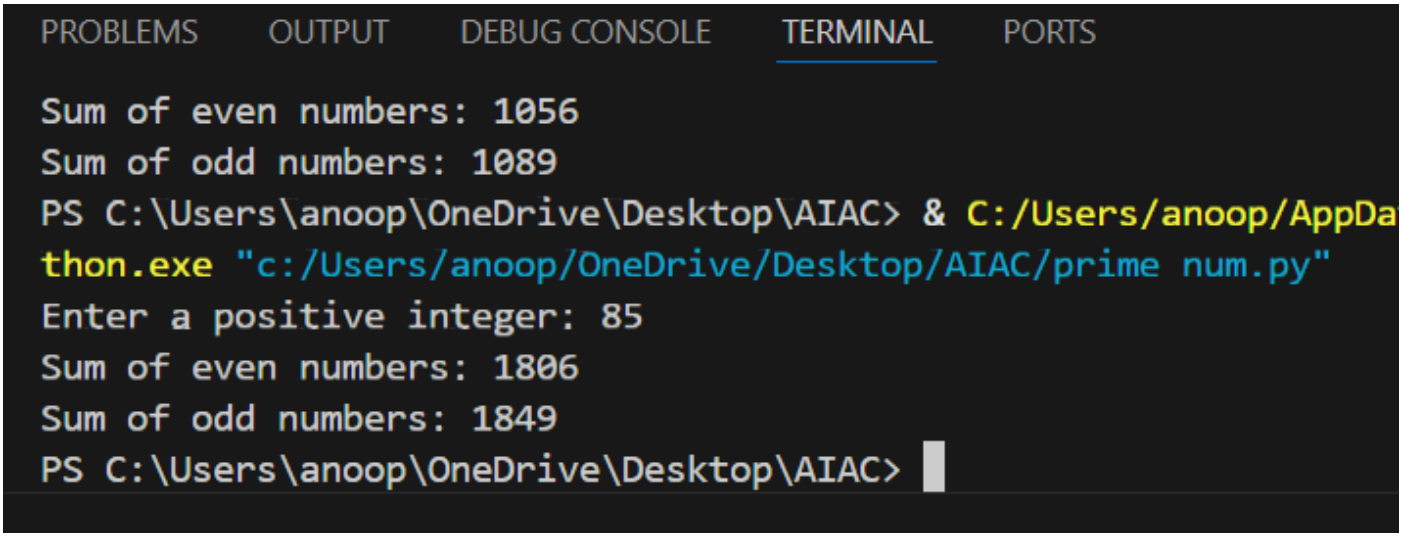
```
    Returns:
        Tuple of (even_sum, odd_sum)
    """
    if num < 1:
        return 0, 0

    even_sum = sum(i for i in range(2, num + 1, 2))
    odd_sum = sum(i for i in range(1, num + 1, 2))
    return even_sum, odd_sum

def get_positive_integer(prompt):
    """Get valid positive integer input from user."""
    while True:
        try:
            value = int(input(prompt))
            if value < 1:
                print("Please enter a positive number.")
                continue
            return value
        except ValueError:
            print("Invalid input. Please enter a valid integer.")

if __name__ == "__main__":
    user_input = get_positive_integer("Enter a positive number: ")
    even, odd = even_odd_sum_tuple(user_input)
    print(f"Sum of even numbers: {even}")
    print(f"Sum of odd numbers: {odd}")
```

output:



Explanation of improvements:

Input validation

- Making sure the user enters a valid integer.
- Handling negative numbers gracefully (return (0,0) or raise an error).

Readability

- Using clear variable names.
- Adding docstrings and comments.

Edge cases

- If input is 0, both sums should be 0.
- If input is negative, we can either reject it or compute sums up to that number (here I'll reject it for clarity).