ASSIGNMENT 2.1

HALL NO:2303A51781

BATCH:28

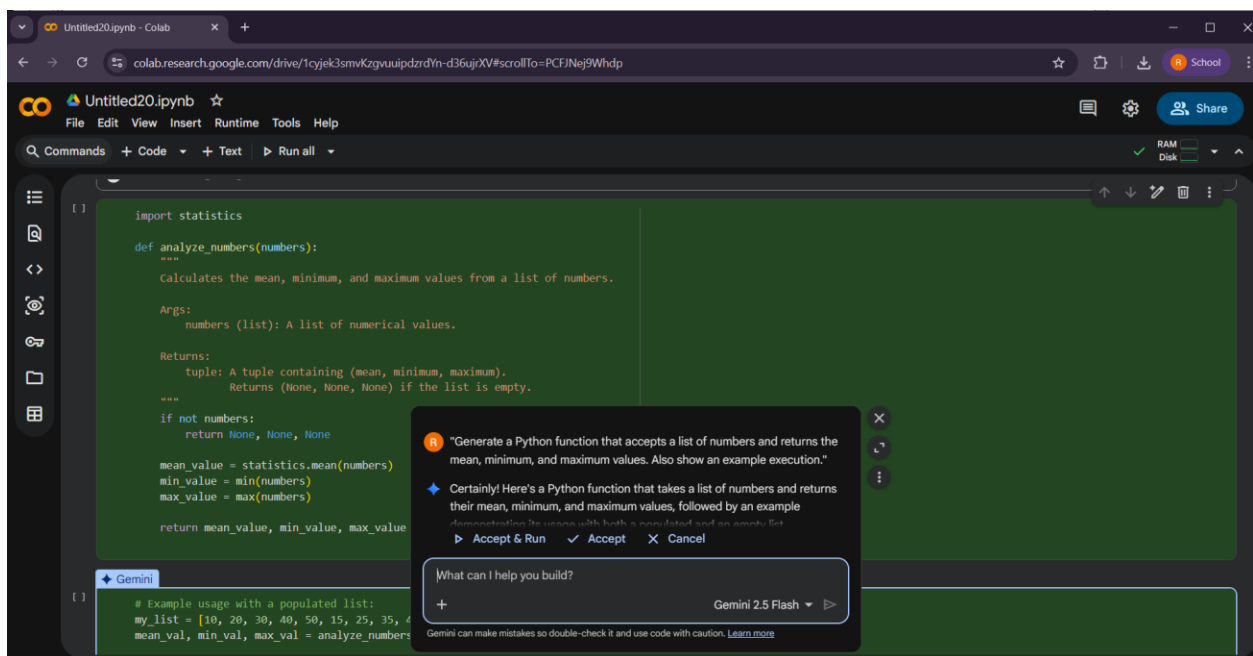## Task 1: Statistical Summary for Survey Data

Task

Generate a Python function that takes a list of numerical survey responses and calculates the mean, minimum, and maximum values.

Prompt (Used in Google Gemini – Colab)

"Generate a Python function that accepts a list of numbers and returns the mean, minimum, and maximum values. Also show an example execution."

Code:

Output:

Justification

This function efficiently computes statistical measures using Python built-in functions. It improves readability, reduces code length, and ensures accurate results, making it suitable for real-world survey data analysis

Task 2: Armstrong Number – AI Comparison

Task:

Generate an Armstrong number checker using Google Gemini and GitHub Copilot, and compare their outputs and clarity.

Prompt (Gemini):

"Write a Python program to check whether a given number is an Armstrong number."

Prompt (GitHub Copilot)

"Create a Python function that validates whether a number is an Armstrong number with clear logic and comments."

Code (Gemini Output):


```python
def is_armstrong_number(number):
    """
    Checks if a given 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.

    Args:
        number (int): The number to check.

    Returns:
        bool: True if the number is an Armstrong number, False otherwise.
    """
    if not isinstance(number, int) or number < 0:
        return False # Armstrong numbers are typically positive integers

    # Convert the number to a string to easily get its digits and length
    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:
```



```python
# Example usage:
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 9474 an Armstrong number? {is_armstrong_number(9474)}") # Expected: True (9^4 + 4^4 + 7^4 + 4^4 = 6561 + 256 + 2401 + 256 = 9474)
print(f"Is 10 an Armstrong number? {is_armstrong_number(10)}")   # Expected: False (1^2 + 0^2 = 1)
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 0 an Armstrong number? {is_armstrong_number(0)}")    # Expected: True (0^1 = 0)
print(f"Is -5 an Armstrong number? {is_armstrong_number(-5)}")   # Expected: False (Negative numbers are typically not considered)
```

```
Is 153 an Armstrong number? True
Is 9474 an Armstrong number? True
Is 10 an Armstrong number? False
Is 1634 an Armstrong number? True
Is 0 an Armstrong number? True
Is -5 an Armstrong number? False
```

```python
# Example usage with a populated list:
my_list = [10, 20, 30, 40, 50, 15, 25, 35, 45]
mean_val, min_val, max_val = analyze_numbers(my_list)

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

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

Original List: [10, 20, 30, 40, 50, 15, 25, 35, 45]
```

Output:

```
Original List: [10, 20, 30, 40, 50, 15, 25, 35, 45]
Mean: 30
Minimum: 10
Maximum: 50

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

Code (Copilot Output):

```python
#Write a Python program to check whether a number is an Armstrong number.
# An Armstrong number (also known as a narcissistic number) is a number that is equal to the sum of its own digits
#  each raised to the power of the number of digits.
def is_armstrong_number(num):
    # Convert the number to string to easily iterate over digits
    str_num = str(num)
    # Get the number of digits
    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
# Example usage:
number = int(input("Enter a number: "))
if is_armstrong_number(number):
    print(f"{number} is an Armstrong number.")
else:
    print(f"{number} is not an Armstrong number.")
```

Output:

```
tivate.ps1"
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding> & "C:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assisant_Coding/.venv/Sc
ripts/python.exe" "c:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assisant_Coding/Assignment(2.1).py"
Enter a number: 153
153 is an Armstrong number.
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
```

Justification: (Comparison between Gemini AI and GitHub Copilot)

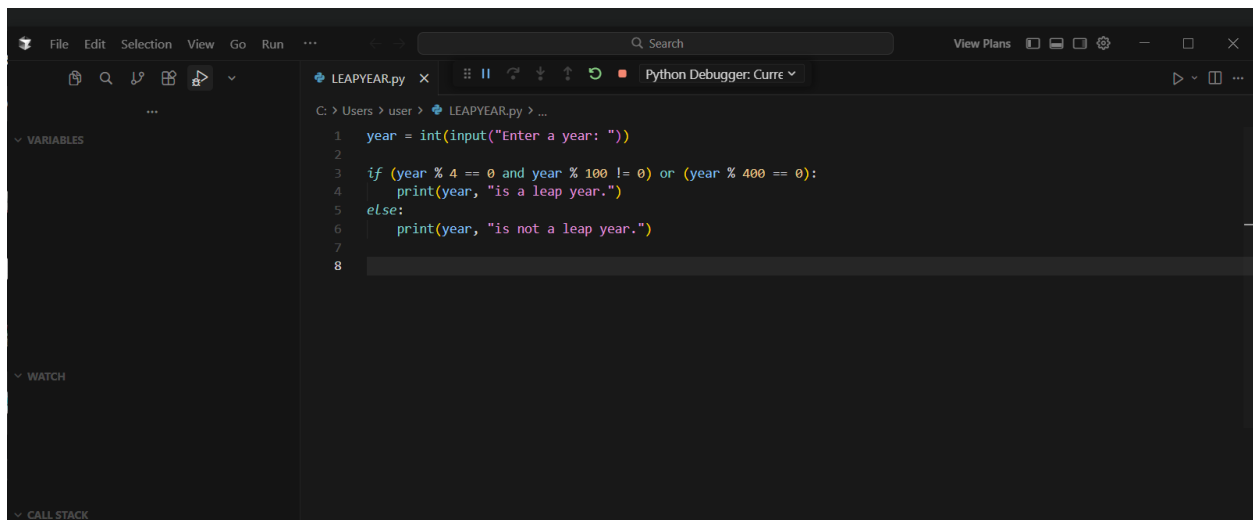| Feature	Gemini (Google Colab)	Copilot (VS Code)
Code clarity	High – code is structured with clear flow and validation	Medium – logic is correct but less structured
Use of functions	Yes – uses a well-defined function with validation	Yes – uses a function but with minimal safeguards
Readability	Good – descriptive variable names and spacing	Good – concise but slightly compact
Comments	More – includes detailed docstring and inline comments	Less – basic comments only
Logic approach	Direct – step-by-step logic easy to follow	Optimized – uses concise expressions (generator)
Error handling	Present – checks input type and negative values	Not present – assumes valid input
Suitability for learning	High – beginner-friendly and explanatory	Medium – better for experienced users

Task 3: Leap Year Validation Using Cursor AI

Task

Generate two versions of a Python program to check leap years using different prompts in Cursor AI.

Prompt 1

"Write a simple Python program to check whether a year is a leap year."


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

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

Output:


```
PS C:\Users\user> & 'c:\Users\user\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\user\.cursor\extensions\ms-python
.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '54056' '--' 'c:\Users\user\LEAPYEAR.py'
Enter a year: 2024
2024 is a leap year.
PS C:\Users\user>
```
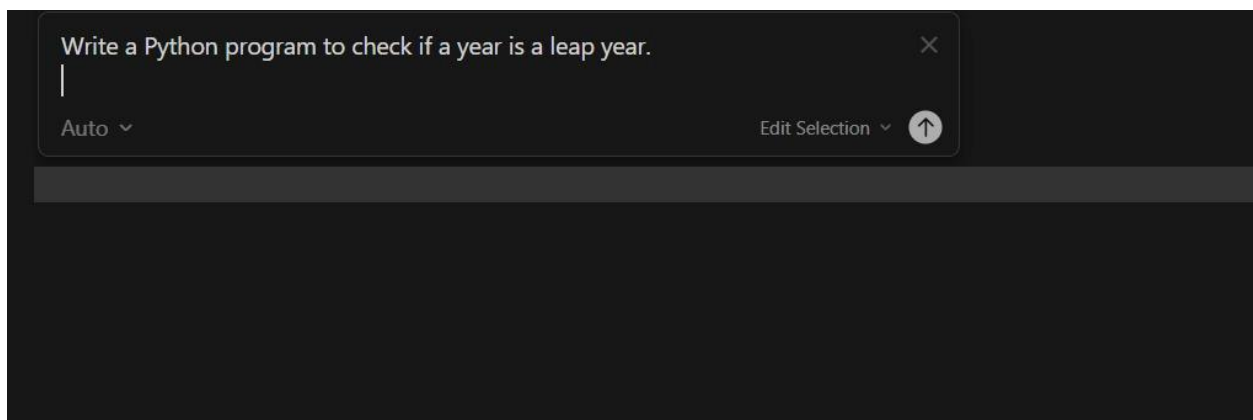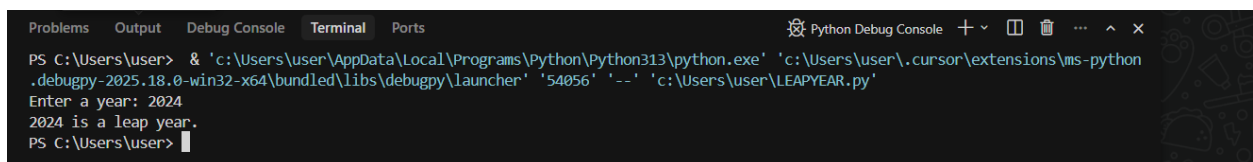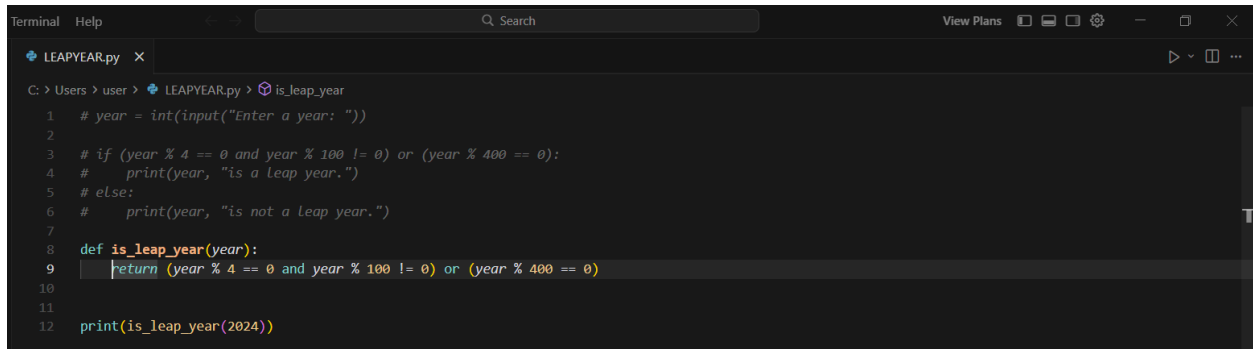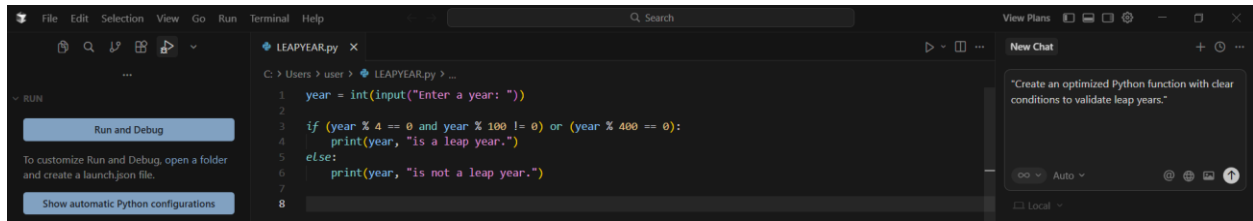
The second version is more modular and reusable. Cursor AI adapts code quality based on prompt specificity, highlighting the importance of precise instructions.

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

Task

Write a Python program to calculate the sum of odd and even numbers in a tuple, then refactor it using AI.

Original Code (Student Logic)

```python
#Adding Student own code on even Or odd number without using AI
#create  a function for sum of even and odd numbers
def sum_even_odd(nums):
    sum_even=0
    sum_odd=0
    for i in nums:
        if i%2==0:
            sum_even+=i
        else:
            sum_odd+=i
    print("sum of Even Numbers: ",sum_even)
    print("sum of Even Numbers: ",sum_odd)
li=[1,2,3,4,5,6]
sum_even_odd(li)
```

Output:

```
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding> & "C:/Users/DELL/OneDrive/Desktop/
RNING COURSES/AI_Assisant_Coding/.venv/Scripts/python.exe" "c:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_As
ant_Coding/Assignment(2.1).py"
sum of Even Numbers:  12
sum of Odd Numbers:  9
```

Refactored Code (AI Assisted)

```python
def sum_even_odd(nums):
    """
    Calculates the sum of even and odd numbers in a list.

    Args:
        nums (list): A list of numbers.

    Returns:
        tuple: A tuple containing (sum_even, sum_odd).
    """
    sum_even = 0
    sum_odd = 0
    for num in nums:
        if num % 2 == 0:
            sum_even += num
        else:
            sum_odd += num
    return sum_even, sum_odd

li = [1, 2, 3, 4, 5, 6]
even_sum, odd_sum = sum_even_odd(li)
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
```

Justification:

1.Both programs correctly separate even and odd numbers using the modulus (% 2) condition.

2.The first code is student-written and directly prints results, making it easy to understand for beginners.

3.The second code improves structure by using a docstring and returning values instead of printing inside the function.

4. Returning values in the optimized code increases reusability and makes it suitable for larger programs.

5.Overall, the first code is good for learning basic logic, while the second code follows better programming practices.