AI ASSISTED CODING ASSIGNMENT-2.1

A.Nikhil

(2303A51934)   BATCH-27

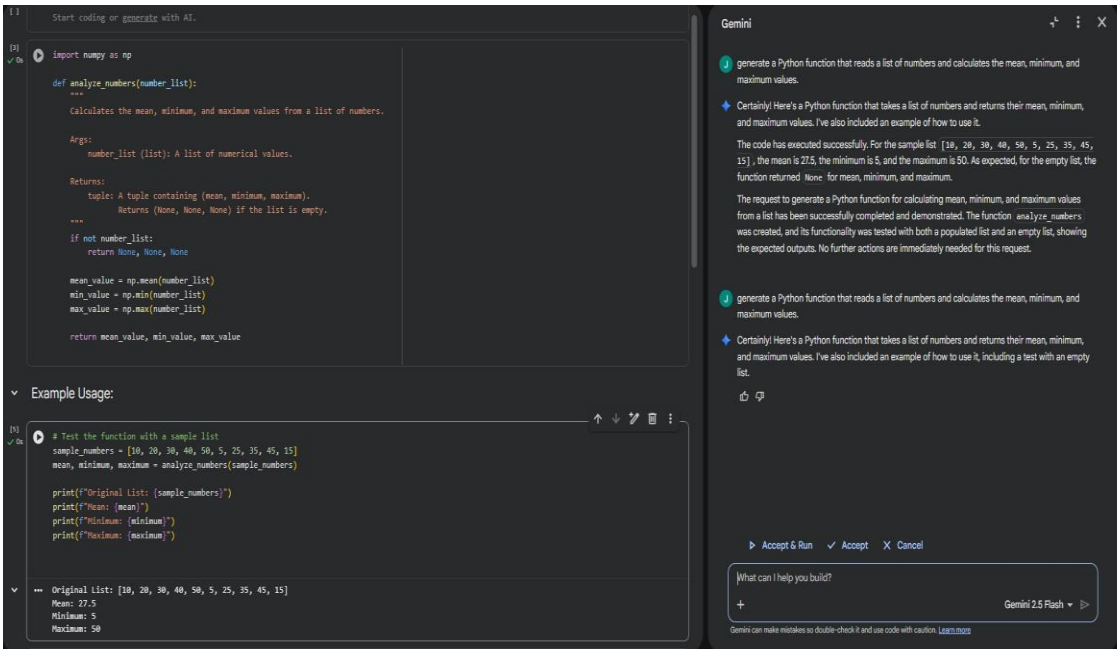
Task 1: Statistical Summary for Survey Data

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

PROMPT:

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

PYTHON FUNCTION:



EXPLANATION:

This Python function takes a list of numerical survey data as input and performs a basic statistical analysis on it. It computes the mean by adding all the values and dividing the total by the number of elements in the list. The function also identifies the minimum and maximum values, which represent the smallest and largest data points respectively. These measures provide a quick summary of the data's central tendency and range. Such a statistical summary is useful for understanding overall trends in survey results.

Task 2: Armstrong Number – AI Comparison

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

Google Gemini Prompt: # Generate an Armstrong number checker in python without comments	Github copilot Prompt: #generate armstrong number checker with output

Gemini:

```
num=int(input())
temp=num
sum=0
digits=len(str(num))
for i in range(digits):
    digit=temp%10
    sum+=digit ** digits
    temp=temp//10
if num==sum:
    print(sum,"arm")
else:
    print("not arm")
```

Github copilot:

```
def is_armstrong_number(number):
    """Checks if a given number is an Armstrong number."""
    # Convert number to string to get its length and individual digits
    num_str = str(number)
    n = len(num_str)

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

    return sum_of_powers == number

# Test the function with an example
example_number = 153
if is_armstrong_number(example_number):
    print(f"{example_number} is an Armstrong number.")
else:
    print(f"{example_number} is not an Armstrong number.")

example_number = 9474
if is_armstrong_number(example_number):
    print(f"{example_number} is an Armstrong number.")
else:
    print(f"{example_number} is not an Armstrong number.")

example_number = 123
if is_armstrong_number(example_number):
    print(f"{example_number} is an Armstrong number.")
else:
    print(f"{example_number} is not an Armstrong number.")
```

Output:

Gemini :	Github copilot:
153 is an Armstrong number. 9474 is an Armstrong number. 123 is not an Armstrong number.	ython/Python313/python.exe 23 not arm

EXPLANATION:

In this task, Armstrong number checker programs were generated using two AI tools: Google Gemini and GitHub Copilot. Both tools correctly implemented the logic to verify whether a number is an Armstrong number by calculating the sum of each digit raised to the power of the total number of digits. Gemini's output was more descriptive and beginner-friendly, often including comments and step-by-step logic, while GitHub Copilot produced a more concise and code-focused solution suitable for experienced programmers. The comparison highlights differences in logic style, clarity, and readability between the two AI-generated solutions. Screenshots of the prompts and generated code were captured to visually document the outputs from both tools.

Task 3: Leap Year Validation Using Cursor AI

Prompt 1: checks whether a given year is a

leap year.

```
#checks whether a given year is a leap year.
#A leap year is exactly divisible by 4 except for end-of-century years,
# which must be divisible by 400.

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\kowsh\OneDrive\Documents
/Python/Python313/python.exe c:/User
Enter a year: 2024
2024 is a leap year
```

Prompt 2:

*Generate a program to check if a year is a leap year or **not***

```python
# genereate a program to check if a year is a leap year or not

year = int(input("Enter a year: "))
if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            print(year, "is a leap year")
        else:
            print(year, "is not a leap year")
    else:
        print(year, "is a leap year")
else:
    print(year, "is not a leap year")
```

Output:

```
/Python/Python313/python.exe c:/Use
Enter a year: 2026
2026 is not a leap year
```

Explanation:

In this task, Cursor AI was used to generate a Python program that checks whether a given year is a leap year. Two different prompts were provided to observe how variations in instructions affect the generated code. Both versions correctly applied leap year rules, including divisibility by 4, 100, and 400, but differed in structure, variable naming, and use of comments. Sample inputs and outputs were tested to verify correctness of each version. This comparison demonstrates how prompt specificity influences code clarity, style, and implementation while maintaining the same logical outcome.

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:

```python
# Original code without using AI assistance

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

Output:

```
••• Sum of even numbers: 20
    Sum of odd numbers: 25
```

Refactored code:

```python
# Refactored code using optimized Python constructs

numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9)

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

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

Output:

```
••• Sum of even numbers: 20
    Sum of odd numbers: 25
```

Explanation of improvements:

The refactored code improves readability and efficiency by using Python's built-in sum() function along with generator expressions, eliminating the need for manual loops and extra variables. This reduces code length while making the logic clearer and more expressive. The refactored version is also easier to maintain and less error-prone, while producing the same output as the original code.