Name : K.Abhinav
HT no: 2303A51856

Assignment: 2

Task 1: Statistical Summary for Survey Data

Prompt : Generate a Python function that accepts a list of numbers and calculates the mean, minimum, and maximum values. Also show example usage and output

```
import statistics

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

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

    Returns:
        dict: A dictionary containing 'mean', 'minimum', and 'maximum' keys.
              Returns None for any value if the list is empty.
    """
    if not numbers:
        return {'mean': None, 'minimum': None, 'maximum': None}

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

    return {'mean': mean_value, 'minimum': min_value, 'maximum': max_value}
```

```
# Example Usage:
my_list = [10, 20, 30, 40, 50, 15, 25, 35]
analysis_results = analyze_numbers(my_list)

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

# Example with an empty list
empty_list = []
empty_results = analyze_numbers(empty_list)
print(f"\nEmpty List: {empty_list}")
print(f"Mean (empty list): {empty_results['mean']}")
print(f"Minimum (empty list): {empty_results['minimum']}")
print(f"Maximum (empty list): {empty_results['maximum']}")
```

Output:

```
Original List: [10, 20, 30, 40, 50, 15, 25, 35]
Mean: 28.125
Minimum: 10
Maximum: 50

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

Observation

- Gemini generated **clean, readable, and beginner-friendly code**
- Included **edge case handling**
- Output was immediately executable in Colab

Justification

Gemini was used because it helps generate Python code easily using simple language.
The prompt asks Gemini to calculate mean, minimum, and maximum values, which are common tasks in data analysis.
Gemini provides clear code with example input and output, making it easy to understand and verify the results in Google Colab.

Task 2:

Armstrong Number – AI Comparison

Prompt: Generate a Python function to check whether a given number is an Armstrong number. Explain the logic briefly and include an example input and output.

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

    An Armstrong number (or narcissistic 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:
        return False
    if number == 0: # 0 is considered an Armstrong number
        return True

    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
```

Gemini

```python
# Example Usage:

# Test cases
number1 = 153
number2 = 370
number3 = 9474
number4 = 123
number5 = 9
number6 = 0

print(f"Is {number1} an Armstrong number? {is_armstrong(number1)}")
print(f"Is {number2} an Armstrong number? {is_armstrong(number2)}")
print(f"Is {number3} an Armstrong number? {is_armstrong(number3)}")
print(f"Is {number4} an Armstrong number? {is_armstrong(number4)}")
print(f"Is {number5} an Armstrong number? {is_armstrong(number5)}")
print(f"Is {number6} an Armstrong number? {is_armstrong(number6)}")
```

Output :

```
Is 153 an Armstrong number? True
Is 370 an Armstrong number? True
Is 9474 an Armstrong number? True
Is 123 an Armstrong number? False
Is 9 an Armstrong number? True
Is 0 an Armstrong number? True
```

Justification

Gemini was used to generate the Armstrong number program because it provides both the Python code and a clear explanation of the logic.
The prompt was designed to check not only code generation but also how well Gemini explains numeric validation concepts.
Including example input and output helps verify the correctness of the program and makes the logic easy to understand.

Task 3:

Leap Year Validation Using Cursor AI.

Prompt: Write a Python program to check whether a given year is a leap year

Code:

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

if (year % 4 == 0):
    if (year % 100 == 0):
        if (year % 400 == 0):
            print("Leap year")
        else:
            print("Not a leap year")
    else:
        print("Leap year")
else:
    print("Not a leap year")
```

Output:

```
Odd sum: 25
Even sum: 30
```

Justification:

Leap year validation ensures accurate date handling in backend systems. Using different prompts in Cursor AI shows how prompt clarity affects code structure—producing either a simple script or a reusable function. This demonstrates the importance of well-defined prompts for generating reliable and maintainable code.

Task 4:

Student Logic + AI Refactoring (Odd/Even Sum)

Prompt: Refactor a Python program that calculates the sum of odd and even numbers in a tuple. Make the code more concise, readable, and Pythonic using best practices, such as built-in functions and efficient structures, while keeping the same functionality and output.

Code:

```
def calculate_odd_even_sums(numbers):
    odd_sum = 0
    even_sum = 0
    for num in numbers:
        if num % 2 == 0:
            even_sum += num
        else:
            odd_sum += num
    return odd_sum, even_sum

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_sum, even_sum = calculate_odd_even_sums(numbers)
print(f"Odd sum: {odd_sum}")
print(f"Even sum: {even_sum}")
```

Output:

```
Odd sum: 25
Even sum: 30
```

Justification:

The AI-refactored code is more concise, readable, and Pythonic. It uses built-in functions and efficient structures, making it easier to maintain and less error-prone while keeping the original functionality intact.