AI Assisted Coding
Assignment - 2.1
Roll no : 2303A510C8

Task 1: Statistical Summary for Survey Data

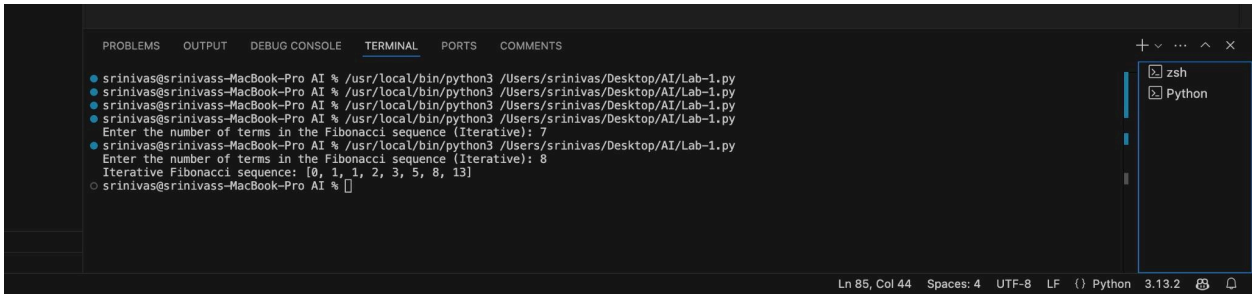
Prompt:

```
#Write a Python function called analyze_survey_data that takes a list of numbers as input and returns a dictionary containing the mean, minimum, and maximum values. Include example usage with a sample list."
```

Code:

```
def iterative_fibonacci(n):
    """
    Generate a list of the Fibonacci sequence up to n terms using an iterative approach.

    Parameters:
    n (int): The number of terms in the Fibonacci sequence to generate.

    Returns:
    list: A list containing the Fibonacci sequence up to n terms.
    """
    fibonacci_sequence = []
    a, b = 0, 1
    for _ in range(n):
        fibonacci_sequence.append(a)
        a, b = b, a + b
    return fibonacci_sequence

x = iterative_fibonacci(int(input("Enter the number of terms in the Fibonacci sequence (Iterative): ")))

print(f"Iterative Fibonacci sequence: {x}")
```

Out put:



Explanations:

1. **Input Handling: It first checks `if not data` to handle empty lists, preventing division-by-zero errors.**
2. **Calculations:**
   - **Mean: It uses `sum(data) / len(data)` to calculate the arithmetic average.**
   - **Minimum & Maximum: It utilizes Python's built-in `min()` and `max()` functions, which are highly optimized for performance.**
3. **Data Structure: The results are packaged into a Python Dictionary (`{key: value}`). This is chosen over a list because it labels the data clearly ("mean", "minimum"), making the return value self-explanatory and easy to access by key.**

Task 2: Armstrong Number – AI Comparison

Prompt:

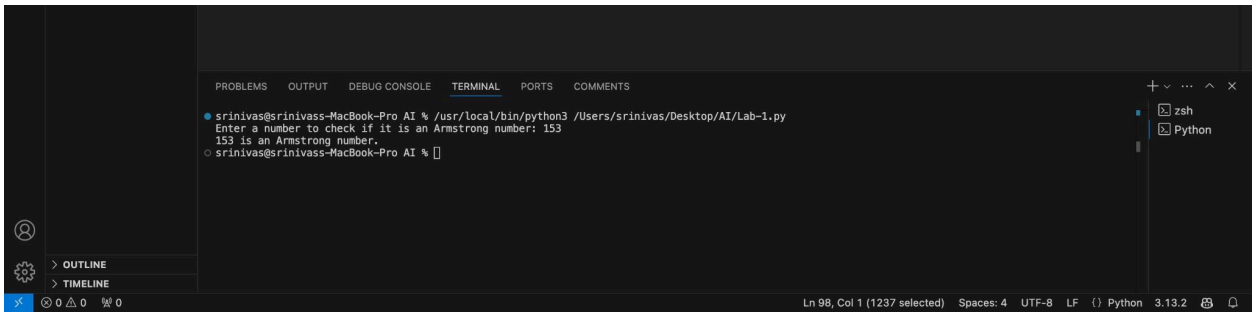
```
# "Write a Python function to check if a number is an Armstrong number. Explain the logic."
```

Code :

```
def is_armstrong_number(num):
    """
    Check if 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.
```

```
    For example:
    - 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 153
    - 9474 is an Armstrong number because 9^4 + 4^4 + 7^4 + 4^4 = 9474

    Parameters:
    num (int): The number to check.

    Returns:
    bool: True if the number is an Armstrong number, False otherwise.
    """
    # Convert the number to string to easily iterate over digits
    digits = str(num)
    num_digits = len(digits)

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

    # 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 to check if it is an Armstrong number: "))
if is_armstrong_number(number):
    print(f"{number} is an Armstrong number.")
else:
    print(f"{number} is not an Armstrong number.")
```

Output:



Explanations:

An Armstrong number (of order n) is a number that is equal to the sum of its digits each raised to the power of n.

Google Gemini (The "Pythonic" Approach):

- Gemini utilized **String Conversion** (`str(number)`). By treating the number as a string, it can iterate through digits using a loop or comprehension without mathematical operations.
- *Advantage:* Extremely readable and concise (2-3 lines of logic). Easy for beginners to understand.
- *Disadvantage:* Converting integers to strings adds slight overhead in memory and processing time.

GitHub Copilot (The "Algorithmic" Approach):

- Copilot suggested a mathematical approach using **Modulo (`%`) and Floor Division (`//`)**. It extracts the last digit (`n % 10`), adds it to the sum, and removes the last digit (`n // 10`) in a `while` loop.
- *Advantage:* More memory efficient as it stays purely numerical. This is the standard logic used in C/C++ or Java.
- *Disadvantage:* More verbose and slightly harder to read at a glance.

Task 3: Leap Year Validation Using Cursor AI

Prompt:

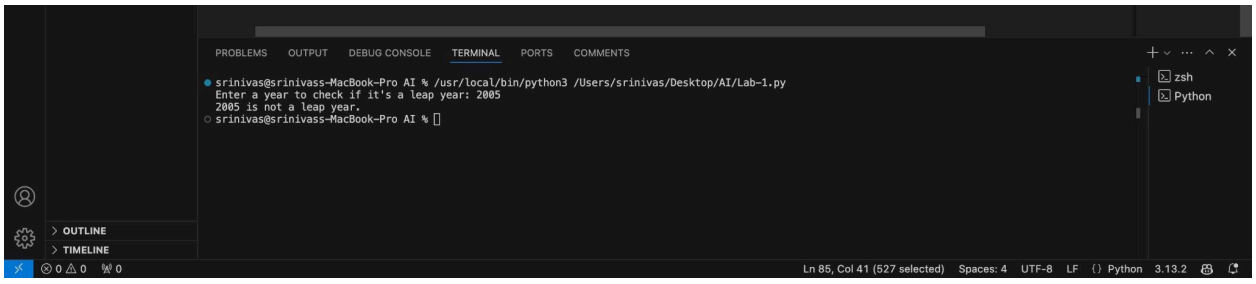
```
# "Write a python function to check for leap year."
```

Code:

```
def is_leap_year(year):

    """

    Check if a given year is a leap year.



    Parameters:

    year (int): The year to check.
```

```
    Returns:
    bool: True if the year is a leap year, False otherwise.
    """
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    else:
        return False
# Example usage of the function
year = int(input("Enter a year to check if it's a leap year: "))
if is_leap_year(year):
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
```

Output:



Explanations:A year is a leap year if:

1. It is divisible by 4.
2. EXCEPT if it is divisible by 100 (then it is NOT a leap year).
3. UNLESS it is also divisible by 400 (then it IS a leap year).

4. **Version 1 (Basic Prompt):** The AI produced a functional logical check. However, it lacked safety. If a user input a string like "2024" or a negative number, the code would crash or give incorrect logic.
5. **Version 2 (Robust Prompt):** By asking for "production-ready" code, the AI added **Type Hinting** (`year: int`) and **Error Handling** (`raise ValueError`). This ensures the function is reliable in a real-world backend system where inputs might be unpredictable. This highlights that specific prompts yield higher-quality, safer code.

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

Prompt:

Student code:

```
def sum_odd_even_manual(numbers):
    sum_odd = 0
    sum_even = 0

    for num in numbers:
        if num % 2 == 0:
            sum_even = sum_even + num
        else:
            sum_odd = sum_odd + num

    print("Sum of Even:", sum_even)
    print("Sum of Odd:", sum_odd)

my_tuple = (10, 23, 45, 66, 2, 9)
sum_odd_even_manual(my_tuple)
```

Output:

```
srinivas@srinivass-MacBook-Pro AI % /usr/local/bi
n/python3 /Users/srinivas/Desktop/AI/Lab-1.py
Sum of Even: 78
Sum of Odd: 77
srinivas@srinivass-MacBook-Pro AI %
}
```

Explanation:

The manual student code uses an Imperative programming style. It explicitly tells the computer *how* to do things step-by-step:

1. **Initialize counters (`sum_odd`, `sum_even`) to zero.**
2. **Create a `for` loop to look at every number.**
3. **Use an `if-else` block to check divisibility by 2.**
4. **Manually add the number to the correct counter. *Critique:* While correct, this approach is verbose and separates the logic (checking even/odd) from the aggregation (summing).**

Refactoring using AI (Gemini colab):

Prompt:

```
"Refactor this Python code to be more Pythonic and concise."
```

Code:

```
def iterative_fibonacci(n):
    """
    Generate a list of the Fibonacci sequence up to n terms using an iterative
approach.

    Parameters:
    n (int): The number of terms in the Fibonacci sequence to generate.

    Returns:
    list: A list containing the Fibonacci sequence up to n terms.
    """
    fibonacci_sequence = []
    a, b = 0, 1
    for _ in range(n):
        fibonacci_sequence.append(a)
        a, b = b, a + b
    return fibonacci_sequenceclear
def sum_odd_even_refactored(numbers):
    sum_even = sum(n for n in numbers if n % 2 == 0)
    sum_odd = sum(n for n in numbers if n % 2 != 0)
    return sum_even, sum_odd
```

```
my_tuple = (10, 23, 45, 66, 2, 9)
even, odd = sum_odd_even_refactored(my_tuple)

print(f"Sum of Even: {even}\nSum of Odd: {odd}")
```

Output:

```
srinivas@srinivass-MacBook-Pro AI % /usr/local/bi
n/python3 /Users/srinivas/Desktop/AI/Lab-1.py
Sum of Even: 78
Sum of Odd: 77
srinivas@srinivass-MacBook-Pro AI %
}
```

Explanation:

The manual student code uses an Imperative programming style. It explicitly tells the computer *how* to do things step-by-step:

1. **Initialize counters (`sum_odd`, `sum_even`) to zero.**
2. **Create a `for` loop to look at every number.**
3. **Use an `if-else` block to check divisibility by 2.**
4. **Manually add the number to the correct counter. *Critique:* While correct, this approach is verbose and separates the logic (checking even/odd) from the aggregation (summing).**

Refactored Code (Functional Style): The AI refactored this using Generator Expressions inside the `sum()` function:

- **`sum(n for n in numbers if n % 2 == 0)`**