

## AI ASSISTED CODING

G.LIKHITHA RAO

2303A52487

BATCH 35

### Assignment-2.1

#### Task-1:

#### 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.

#### **Prompt:**

Generate a Python function that takes a list of numbers as input and calculates the mean, minimum, and maximum values. Also show sample output.

#### **Code:**

```
[3] Os ▶ def calculate_stats(numbers):
        """
        Calculates the mean, minimum, and maximum of a list of numbers.
        """
        if not numbers:
            return None, None, None

        min_val = min(numbers)
        max_val = max(numbers)
        mean_val = sum(numbers) / len(numbers)

        return mean_val, min_val, max_val

# Example usage:
data = [10, 20, 30, 40, 50]
mean, minimum, maximum = calculate_stats(data)

print(f"Mean: {mean}")
print(f"Minimum: {minimum}")
print(f"Maximum: {maximum}")
```

## Output:

```
... Mean: 30.0
    Minimum: 10
    Maximum: 50
```

## Explanation:

This code summarizes a list of numbers by identifying its boundaries and its center.

- **Boundary Detection:** The `min()` and `max()` functions scan the entire list to find the smallest and largest values.
- **Mean Calculation:** The code uses `sum(numbers) / len(numbers)` to divide the total of all values by the count of items in the list.
- **Crash Prevention:** The `if not numbers` check ensures the code returns `None` instead of crashing if an empty list is provided.
- **Result Packing:** It returns all three values at once, which are then "unpacked" into the variables `mean`, `minimum`, and `maximum` for easy printing.

## 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

### Gemini Prompt:

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

### Code:

```
def is_armstrong(number):  
    # Convert number to string to easily iterate through digits  
    digits = str(number)  
    num_digits = len(digits)  
  
    # Calculate the sum of digits raised to the power of num_digits  
    total_sum = sum(int(digit) ** num_digits for digit in digits)  
  
    return total_sum == number  
  
# Test the function  
num = int(input("Enter a number: "))  
if is_armstrong(num):  
    print(f"{num} is an Armstrong number.")  
else:  
    print(f"{num} is not an Armstrong number.")
```



### Output:

```
... Enter a number: 153  
153 is an Armstrong number.
```

### GitHub Copilot:

#Armstrong Number Checker

## Code:

```
armstrong.py >  armstrong.py
1   #Armstrong.number.checker
2  def is_armstrong_number(number):
3      # Convert the number to string to easily iterate over digits
4      digits = str(number)
5      num_digits = len(digits)
6
7      # Calculate the sum of each digit raised to the power of num_digits
8      sum_of_powers = sum(int(digit) ** num_digits for digit in digits)
9
10     # An Armstrong number is one where the sum of the powers equals the original number
11     return sum_of_powers == number
12 # Example usage
13 number = 153
14 if is_armstrong_number(number):
15     print(f"{number} is an Armstrong number.")
16 else:
17     print(f"{number} is not an Armstrong number.")
18 # Output:
19 # 153 is an Armstrong number.
```



## Output:

Example usage



153 is an Armstrong number.

## Comparison table:

AI Coding Tools: A Comparison			
Feature	 Google Gemini AI		 Google Colab
Purpose	AI assistant for code generation, and problem solving	Cloud-based platform to write and execute code	Cloud-based platform to write and execute Python code
Code Generation	Generates code using natural language prompts	Does not generate code by itself	Does not generate code by itself
Code Explanation	Explains logic of code prompts	No built-in explanation feature	No built-in explanation feature
Execution of Code	Prompt-based (chat-style interaction)	Cannot execute code	Notebook-based (cells for code and output)
Error Detection	Suggests fixes and improvements	Medium - requires coding knowledge	Shows runtime and syntax errors
Best Use Case	Understanding logic, generating code, learning concepts	Running programs, testing outputs, data analysis	Supports Python libraries like NumPy, Pandas, Matplotlib

### 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

### Prompt 1:

Write a Python program to check whether a given year is a leap year.

### Code:

```
def is_leap_year(year: int) -> bool:
    # Leap year if divisible by 4, except years divisible by 100,
    # unless also divisible by 400.
    return (year % 4 == 0) and (year % 100 != 0 or year % 400 == 0)

if __name__ == "__main__":
    try:
        year = int(input("Enter a year: ").strip())
        print(f"{year} is a leap year." if is_leap_year(year) else f"{year} is r
    except ValueError:
        print("Please enter a valid integer year.")
```

### Output:

```
... Enter a year: 2025
      2025 is not a leap year.
```

### Prompt 2:

Generate a well-commented Python program to validate leap year with input and output examples.

### Code:



```
def is_leap_year(year: int) -> bool:
    """
    Return True if the given Gregorian calendar year is a leap year.

    Rules:
    - Leap if divisible by 4,
    - except years divisible by 100,
    - unless divisible by 400.
    """
    # Divisible by 400 => leap (covers years like 2000)
    if year % 400 == 0:
        return True
    # Divisible by 100 (but not 400) => common year (covers years like 1900)
    if year % 100 == 0:
        return False
    # Divisible by 4 (but not 100) => leap (covers years like 2024)
    if year % 4 == 0:
        return True
    # Otherwise => common year
    return False

def main():
    # Prompt the user; strip to allow clean parsing, then convert to int
    raw = input("Enter a year: ").strip()
    try:
        year = int(raw)
    except ValueError:
        print(f"Invalid year: {raw!r}. Please enter an integer, e.g., 2024.")
        return



    if is_leap_year(year):
        print(f"{year} is a leap year.")
    else:
        print(f"{year} is NOT a leap year.")

if __name__ == "__main__":
    main()
```

## Output:

```
... Enter a year: 2021
    2021 is NOT a leap year.
```

### Task 3: Comparison Between Two Prompts (Cursor AI – Leap Year Program)

Aspect	Prompt 1	Prompt 2	
Prompt Used	<i>"Write a Python program to check whether a given year is a leap year."</i>	<i>"Generate a well-commented Python program to validate leap year with sample input and output."</i>	
Input Method	Takes year as input from the user	Takes year as input from the user	
Code Structure	Simple and concise	Structured and well-organized	
Use of Comments	Minimal or no comments	Detailed comments explaining each step	
Readability	Easy to read	Very easy to understand	

### 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

**My code(Input&Output):**



5

```
▶ t = (1, 2, 3, 4, 5, 6)
  even_sum = 0
  odd_sum = 0

  for i in t:
      if i % 2 == 0:
          even_sum += i
      else:
          odd_sum += i

  print("Even Sum:", even_sum)
  print("Odd Sum:", odd_sum)
```

```
... Even Sum: 12
    Odd Sum: 9
```

### Refactored Code(Input&Output):

```
▶ numbers_tuple = (1, 2, 3, 4, 5, 6)

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

  print(f"Refactored Code Output:")
  print(f"Even Sum: {even_sum}")
  print(f"Odd Sum: {odd_sum}")
```

```
... Refactored Code Output:
    Even Sum: 12
    Odd Sum: 9
```

### Improvements in AI-Refactored Code:

1. The refactored code reduces the number of lines by using Python's built-in `sum()` function.
2. Generator expressions replace manual loops, making the code more efficient.
3. Readability is improved by using concise and expressive syntax.
4. The logic becomes easier to maintain and modify in future.
5. The output remains the same while the code quality is enhanced.