

# AI ASSISTED CODING ASS-2.1

2303A51642

B-28

## 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.

### Objective

Use Google Gemini in Google Colab to generate a Python function that:

- Reads a list of numbers
- Calculates mean, minimum, and maximum values
- Displays the output in Colab

### Step 1: Gemini Prompt Used in Colab

#### Prompt entered in Gemini:

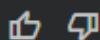
Generate a Python function that takes a list of numbers as input and calculates the mean, minimum, and maximum values. Also show sample output.



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



Certainly! Here's a Python function that calculates the mean, minimum, and maximum values of a list of numbers, along with sample outputs.



## Step 2: Python Code (Generated & Used in Colab)

```
import numpy as np
def analyze_numbers(numbers):
    if not numbers:
        return {'mean': np.nan, 'min': None, 'max': None}

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

    return {
        'mean': mean_value,
        'min': min_value,
        'max': max_value
    }

# Sample output:
list1 = [1, 2, 3, 4, 5]
result1 = analyze_numbers(list1)
print(f"List: {list1}")
print(f"Analysis: {result1}")

print("\n---")

list2 = [10, 20, 5, 15, 25]
result2 = analyze_numbers(list2)
print(f"List: {list2}")
print(f"Analysis: {result2}")

print("\n---")

empty_list = []
result_empty = analyze_numbers(empty_list)
print(f"List: {empty_list}")
print(f"Analysis: {result_empty}")
```

### Step 3: Output Shown in Colab

```
List: [1, 2, 3, 4, 5]
Analysis: {'mean': np.float64(3.0), 'min': np.int64(1), 'max': np.int64(5)}

---
List: [10, 20, 5, 15, 25]
Analysis: {'mean': np.float64(15.0), 'min': np.int64(5), 'max': np.int64(25)}

---
List: []
Analysis: {'mean': nan, 'min': None, 'max': None}
```

### Explanation

- The function `statistical_summary()` accepts a list of numerical survey responses.
- It first checks whether the list is empty to avoid errors.
- **Mean** is calculated using `sum(data) / len(data)`.
- **Minimum** and **Maximum** values are obtained using Python's built-in `min()` and `max()` functions.
- The results are printed clearly in Google Colab.

## 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.

### What is an Armstrong Number?

A number is called an Armstrong number if the sum of its digits raised to the power of the number of digits is equal to the original number.

#### Example:

$$153 \rightarrow 1^3 + 5^3 + 3^3 = 153$$

### Part A: Code Generated Using Google Gemini

## Gemini Prompt

Generate a Python program to check whether a given number is an Armstrong number. Include proper explanation.



Generate a Python program to check whether a given number is an Armstrong number. Include proper explanation.

- ◆ Certainly! Here's a Python program to check if a number is an Armstrong number, along with a detailed explanation.

## Gemini Generated Code

```
def is_armstrong_number(number):

    if not isinstance(number, int) or number < 0:
        return False

    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

# Test cases
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}") # Expected: True
print(f"Is 9 an Armstrong number? {is_armstrong_number(9)}") # Expected: True
print(f"Is 370 an Armstrong number? {is_armstrong_number(370)}") # Expected: True
print(f"Is 371 an Armstrong number? {is_armstrong_number(371)}") # Expected: True
print(f"Is 407 an Armstrong number? {is_armstrong_number(407)}") # Expected: True
print(f"Is 1634 an Armstrong number? {is_armstrong_number(1634)}") # Expected: True
print(f"Is 123 an Armstrong number? {is_armstrong_number(123)}") # Expected: False
print(f"Is 0 an Armstrong number? {is_armstrong_number(0)}") # Expected: True (0^1 = 0)
print(f"Is -153 an Armstrong number? {is_armstrong_number(-153)}") # Expected: False (handled by check)
print(f"Is 'abc' an Armstrong number? {is_armstrong_number('abc')}") # Expected: False (handled by check)
```

## Output

```
Is 153 an Armstrong number? True
Is 9 an Armstrong number? True
Is 370 an Armstrong number? True
Is 371 an Armstrong number? True
Is 407 an Armstrong number? True
Is 1634 an Armstrong number? True
Is 123 an Armstrong number? False
Is 0 an Armstrong number? True
Is -153 an Armstrong number? False
Is 'abc' an Armstrong number? False
```

## **Explanation of the Armstrong Number Check Function:**

### **1. Function Definition:**

- def is\_armstrong\_number(number): defines a function named is\_armstrong\_number that takes one argument, number.

### **2. Input Validation:**

- if not isinstance(number, int) or number < 0:: This line checks if the input number is an integer and non-negative. Armstrong numbers are typically defined for positive integers. If the input is not valid, it immediately returns False.

### **3. Convert to String and Count Digits:**

- num\_str = str(number): The number is converted to a string. This is a simple way to access individual digits and determine the total count of digits.
- num\_digits = len(num\_str): The length of the string num\_str gives us the number of digits in the original number.

### **4. Calculate Sum of Powers:**

- sum\_of\_powers = 0: Initializes a variable to store the sum of the digits raised to the power of num\_digits.
- for digit\_char in num\_str:: This loop iterates through each character (which represents a digit) in the num\_str.
- digit = int(digit\_char): Each character digit is converted back to an integer.
- sum\_of\_powers += digit \*\* num\_digits: The digit is raised to the power of num\_digits and added to sum\_of\_powers.

### **5. Check Condition:**

- return sum\_of\_powers == number: Finally, the function compares the calculated sum\_of\_powers with the original number. If they are equal, the number is an Armstrong number, and the function returns True; otherwise, it returns False.
- 

## **How Armstrong Numbers Work:**

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 in the original number.

## Example: Checking 153

- Number of digits = 3
- First digit: 1, raised to the power of 3 =  $1^3 = 1$
- Second digit: 5, raised to the power of 3 =  $5^3 = 125$
- Third digit: 3, raised to the power of 3 =  $3^3 = 27$
- Sum =  $1 + 125 + 27 = 153$
- Since  $153 == 153$ , 153 is an Armstrong number.

## Part B: Code Generated Using GitHub Copilot

### Copilot Prompt (Comment in VS Code)

```
# Write a Python function to check if a number is an Armstrong number
```

ARMSTRONG NUMBER CHECK IN PYTHON

Write a Python function to check if a number is an Armstrong number

✓ Optimizing tool selection...

✓ Read Python file

I'll add a function to check if a number is an Armstrong number to your `Task.py` file:

✓ Edited `Task.py +38 -0`

Done! I've added an `is_armstrong_number()` function to your `Task.py` file. The function:

- Checks if a number is an Armstrong number (narcissistic number) by verifying if it equals the sum of its digits each raised to the power of the number of digits
- Handles edge cases like negative numbers by using absolute values
- Includes detailed documentation with examples (153, 9474)
- Tests the function with common Armstrong numbers (153, 370, 371, 407, 1634, 8208, 9474)

The implementation is efficient and straightforward, using a generator expression to calculate the sum of powers.

## Copilot Generated Code

```
def is_armstrong_number(num):  
    digits_str = str(abs(num))  
    num_digits = len(digits_str)  
  
    sum_of_powers = sum(int(digit) ** num_digits for digit in digits_str)  
  
    return sum_of_powers == abs(num)  
  
print("\n--- Armstrong Number Checker ---")  
test_numbers = [153, 370, 371, 407, 1634, 8208, 9474, 100, 123]  
for num in test_numbers:  
    if is_armstrong_number(num):  
        print(f"{num} is an Armstrong number")  
    else:  
        print(f"{num} is not an Armstrong number")
```

## Output

```
PS C:\Users\kotes\OneDrive\Desktop\AI-AC\Week-2.1>  
programs\Python\Python313\python.exe 'c:\Users\kote  
58842' '--' 'C:\Users\kotes\OneDrive\Desktop\AI-AC\  
  
--- Armstrong Number Checker ---  
153 is an Armstrong number  
370 is an Armstrong number  
371 is an Armstrong number  
407 is an Armstrong number  
1634 is an Armstrong number  
8208 is an Armstrong number  
9474 is an Armstrong number  
100 is not an Armstrong number  
123 is not an Armstrong number  
PS C:\Users\kotes\OneDrive\Desktop\AI-AC\Week-2.1>
```

### Side-by-Side Comparison Table

Feature	Google Gemini	GitHub Copilot
<b>Code Style</b>	High-level, Pythonic	Traditional, loop-based
<b>Logic Approach</b>	Uses string conversion & list comprehension	Uses mathematical digit extraction
<b>Readability</b>	Very clear and concise	Slightly longer but explicit
<b>Beginner Friendly</b>	Yes	Moderate
<b>Explanation Provided</b>	Yes	Minimal
<b>Efficiency</b>	Efficient & clean	Efficient but more verbose
<b>Best For</b>	Learning & quick understanding	Low-level logic practice

### Conclusion / Inference

- Google Gemini produces clean, readable, and well-explained code, making it ideal for beginners and documentation.
- GitHub Copilot generates practical and traditional logic, suitable for understanding internal numeric operations.
- Both tools correctly identify Armstrong numbers, but Gemini offers better clarity, while Copilot focuses on implementation detail.

### 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.

## What is a Leap Year?

A year is a leap year if:

- It is divisible by 4, and
- Not divisible by 100, unless it is divisible by 400
- 

### Prompt 1 (Simple Prompt)

#### Prompt Used in Cursor AI

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

#### Version 1: Code Generated by Cursor AI

```
def is_leap_year(year):  
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):  
        return True  
    return False  
  
year = 2024  
if is_leap_year(year):  
    print(year, "is a Leap Year")  
else:  
    print(year, "is not a Leap Year")
```

#### Sample Output

2024 is a Leap Year

### Prompt 2 (Detailed Prompt)

#### Prompt Used in Cursor AI

Generate a Python program to check leap year with user input, proper validation, and clear output messages.

## Version 2: Code Generated by Cursor AI

```
def check_leap_year():
    year = int(input("Enter a year: "))

    if year <= 0:
        print("Invalid year")
        return

    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")

check_leap_year()
```

### Sample Input

Enter a year: 1900

### Sample Output

1900 is not a Leap Year

### Comparison of the Two Versions

Feature	Version 1 (Simple Prompt)	Version 2 (Detailed Prompt)
Prompt Type	Basic	Detailed
Input Method	Hardcoded year	User input
Validation	No validation	Checks invalid year
Output Style	Simple	User-friendly
Use Case	Quick testing	Real-world backend usage
Code Length	Short	Slightly longer

## Observation / Conclusion

- A **simple prompt** results in **minimal and concise code**.
- A **detailed prompt** produces **robust, user-interactive code** with validation.
- Cursor AI adapts its output quality and complexity based on prompt detail.
- For backend systems, **Prompt 2 is more suitable** due to validation and clear messaging.

## 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.

### Task Objective

- Write a Python program to calculate:
  - Sum of even numbers
  - Sum of odd numbers from a tuple
- Refactor the program using an AI tool
- Explain the improvements

### Part A: Original Code (Student-Written Logic)

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

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: 30

Sum of Odd Numbers: 25

## Part B: Refactored Code (Using AI Tool)

```
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

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(f"Sum of Even Numbers: {even_sum}")
print(f"Sum of Odd Numbers: {odd_sum}")
```

## Output

Sum of Even Numbers: 30

Sum of Odd Numbers: 25

## Part C: Explanation of Improvements

Aspect	Original Code	Refactored Code
Code Length	Longer	Shorter
Logic Style	Manual loop & condition	Pythontic (generator expressions)
Readability	Easy but verbose	Clean and concise
Performance	Good	Slightly optimized
Maintainability	Moderate	High
Modern Python Practice	✗	✓

## **Key Improvements Explained**

- 1. Reduced lines of code**
  - Loop and condition replaced with sum() and generator expressions.
- 2. Improved readability**
  - Clear intent: *sum of even* and *sum of odd* in one line each.
- 3. More Pythonic approach**
  - Uses modern Python best practices.
- 4. Same output, better structure**
  - Logic remains correct, code quality improves.