

Name: P.Sai Venkatesh

H.No: 2303A53039 - B46

Assignment_2.1

Lab 2: Exploring Additional AI Coding Tools beyond Copilot – Gemini (Colab) and Cursor AI

Lab Objectives:

- ❖ To explore and evaluate the functionality of Google Gemini for AI-assisted coding within Google Colab.
- ❖ To understand and use Cursor AI for code generation, explanation, and refactoring.
- ❖ To compare outputs and usability between Gemini, GitHub Copilot, and Cursor AI.
- ❖ To perform code optimization and documentation using AI tools.

Lab Outcomes (LOs):

After completing this lab, students will be able to:

- ❖ Generate Python code using Google Gemini in Google Colab.
- ❖ Analyze the effectiveness of code explanations and suggestions by Gemini.

- ❖ Set up and use Cursor AI for AI-powered coding assistance.
- ❖ Evaluate and refactor code using Cursor AI features.
- ❖ Compare AI tool behavior and code quality across different platforms.

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.

❖ Expected Output:

- Correct Python function
- Output shown in Colab
- Screenshot of Gemini prompt and result

Prompt:

```
#to generate a python function that reads a list of numbers and calculates the mean, minimum, and maximum values.
```

Code:

```
import statistics
```

```

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

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

        Returns:
        tuple: A tuple containing (mean, minimum, maximum).
        Returns (None, None, None) if the list is empty.
    """
    if not numbers:
        return None, None, None

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

    return mean_value, min_value, max_value

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

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

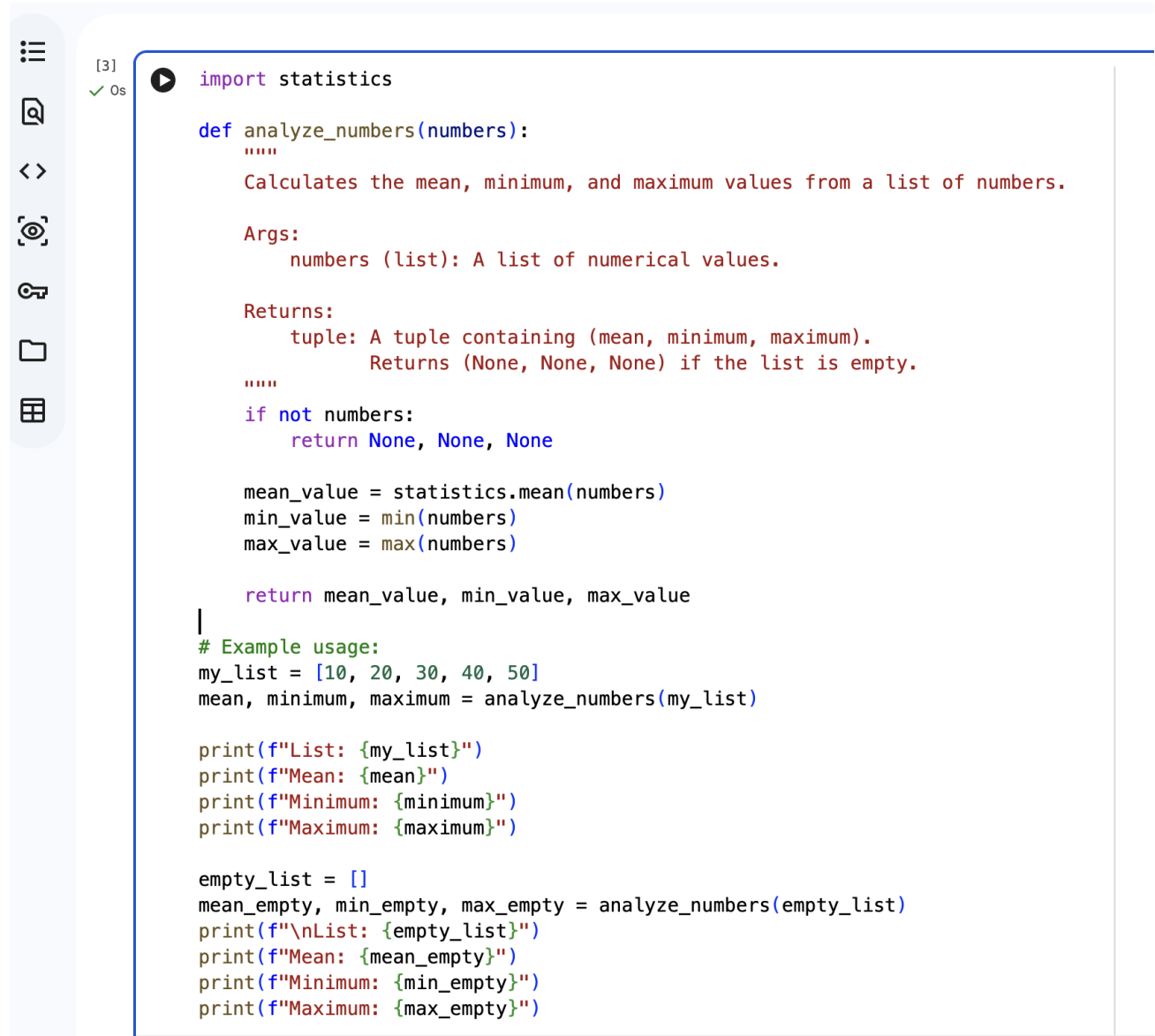
empty_list = []
mean_empty, min_empty, max_empty = analyze_numbers(empty_list)
print(f"\nList: {empty_list}")
print(f"Mean: {mean_empty}")
print(f"Minimum: {min_empty}")
print(f"Maximum: {max_empty}")

```

Output:

List: [10, 20, 30, 40, 50]

Mean: 30
Minimum: 10
Maximum: 50
List: []
Mean: None
Minimum: None
Maximum: None



```
[3]
✓ Os
import statistics

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

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

    Returns:
        tuple: A tuple containing (mean, minimum, maximum).
               Returns (None, None, None) if the list is empty.
    """
    if not numbers:
        return None, None, None

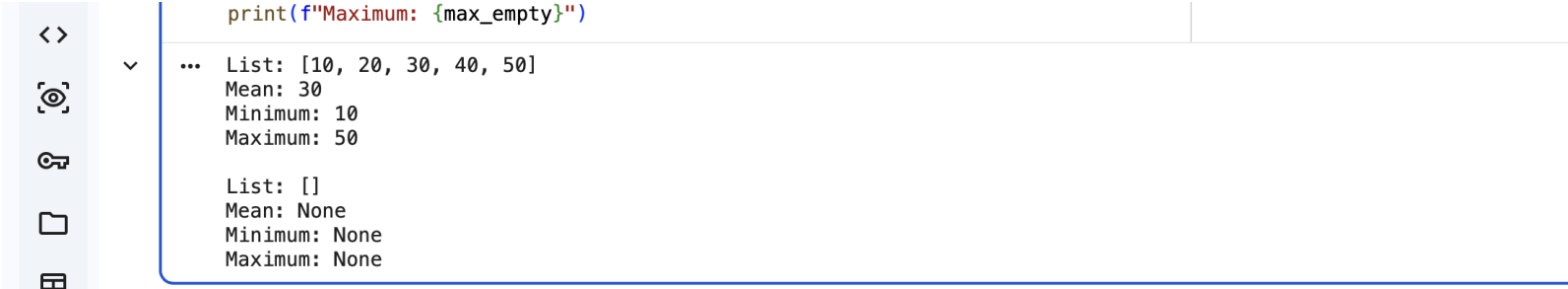
    mean_value = statistics.mean(numbers)
    min_value = min(numbers)
    max_value = max(numbers)

    return mean_value, min_value, max_value

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

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

empty_list = []
mean_empty, min_empty, max_empty = analyze_numbers(empty_list)
print(f"\nList: {empty_list}")
print(f"Mean: {mean_empty}")
print(f"Minimum: {min_empty}")
print(f"Maximum: {max_empty}")
```

A screenshot of a Google Colab interface. On the left is a vertical toolbar with icons for code editor, output, help, file explorer, and runtime. The main area shows a code cell with the line `print(f"Maximum: {max_empty}")`. Below it is an output cell containing two blocks of text. The first block shows the results for a non-empty list: `List: [10, 20, 30, 40, 50]`, `Mean: 30`, `Minimum: 10`, and `Maximum: 50`. The second block shows the results for an empty list: `List: []`, `Mean: None`, `Minimum: None`, and `Maximum: None`.

```
print(f"Maximum: {max_empty}")
```

```
... List: [10, 20, 30, 40, 50]
    Mean: 30
    Minimum: 10
    Maximum: 50

    List: []
    Mean: None
    Minimum: None
    Maximum: None
```

Explanation:

Gemini generated a correct Python function to calculate the mean, minimum, and maximum values from a list.

The output in Colab was accurate and clearly displayed.

Overall, Gemini made the task easy by providing clear and usable code quickly.

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

Prompt:

```
#Generate a python program to check whether a given number is an Armstrong number.
```

Code:

```
def is_armstrong_number(number):  
    """  
        Checks if a given number is an Armstrong number.  
  
        An Armstrong number (also known as a 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 number to check.  
  
        Returns:  
        bool: True if the number is an Armstrong number, False otherwise.  
    """  
    if not isinstance(number, int) or number < 0:  
        return False # Armstrong numbers are typically positive integers  
  
    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  
  
# Example usage:
```

```

print(f"Is 9 an Armstrong number? {is_armstrong_number(9)}") #  $9^1 = 9$ 
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}") #  $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$ 
print(f"Is 370 an Armstrong number? {is_armstrong_number(370)}") #  $3^3 + 7^3 + 0^3 = 27 + 343 + 0 = 370$ 
print(f"Is 371 an Armstrong number? {is_armstrong_number(371)}") #  $3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371$ 
print(f"Is 407 an Armstrong number? {is_armstrong_number(407)}") #  $4^3 + 0^3 + 7^3 = 64 + 0 + 343 = 407$ 
print(f"Is 1634 an Armstrong number? {is_armstrong_number(1634)}") #  $1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634$ 
print(f"Is 123 an Armstrong number? {is_armstrong_number(123)}") # Not an Armstrong number
print(f"Is -153 an Armstrong number? {is_armstrong_number(-153)}")
print(f"Is 0 an Armstrong number? {is_armstrong_number(0)}")

```

Output:

```

Is 9 an Armstrong number? True
Is 153 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 -153 an Armstrong number? False
Is 0 an Armstrong number? True

```

Code Version 2:

```

def is_armstrong_number(number):
    """

```

Checks if a given number is an Armstrong number (another version).

An Armstrong number (also known as a 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 number to check.

Returns:

bool: True if the number is an Armstrong number, False otherwise.

"""

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

```
original_number = number  
num_digits = len(str(number))  
sum_of_powers = 0
```

```
while number > 0:  
    digit = number % 10  
    sum_of_powers += digit ** num_digits  
    number //= 10
```

```
return sum_of_powers == original_number
```

```
# Get input from the user
```

```
try:  
    user_input = int(input("Enter a number to check if it's an Armstrong number: "))  
    if is_armstrong_number(user_input):  
        print(f"{user_input} is an Armstrong number.")  
    else:  
        print(f"{user_input} is not an Armstrong number.")  
except ValueError:  
    print("Invalid input. Please enter an integer.")
```

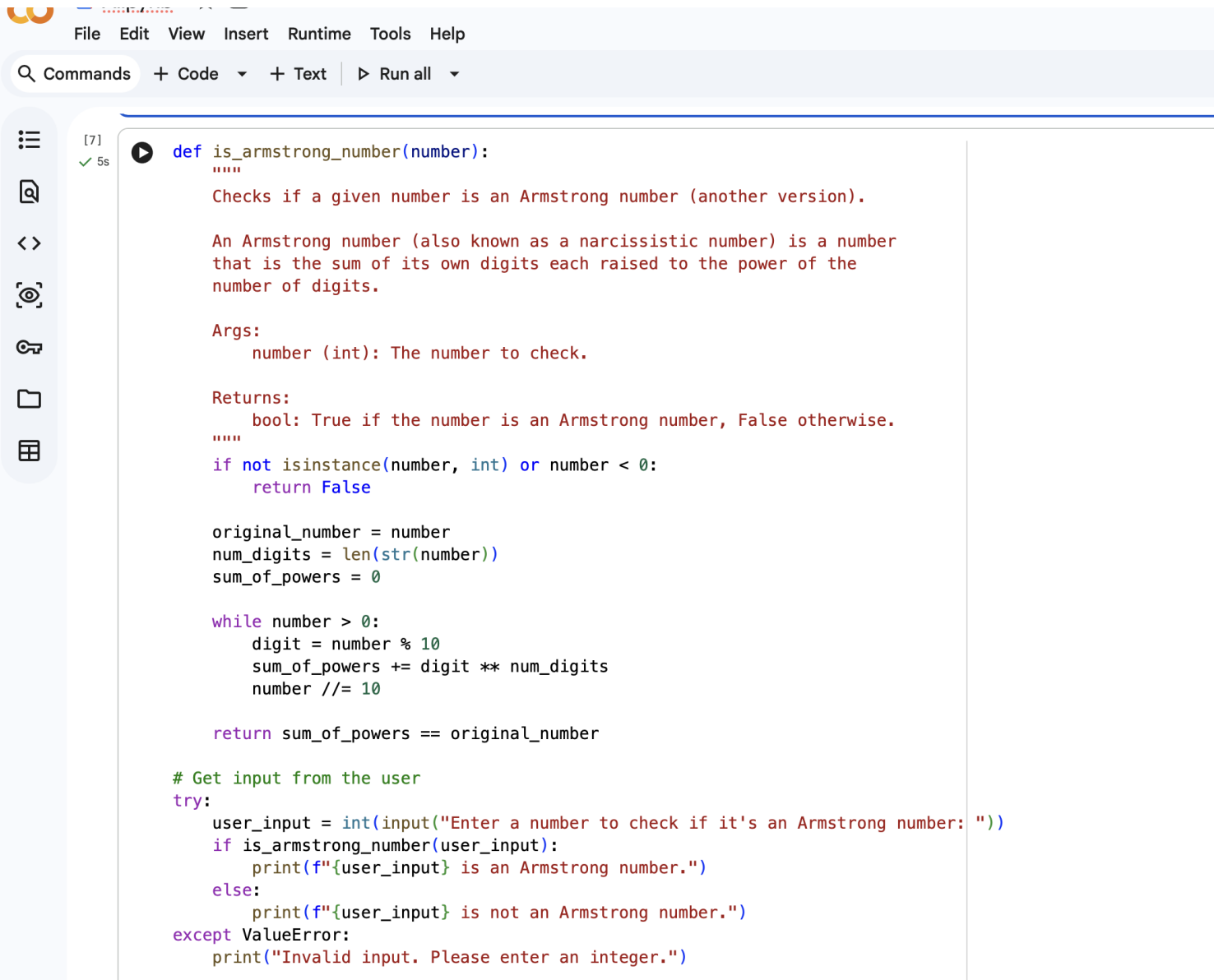
```
# Original example usage (optional, can be removed if desired):  
# print(f"Is 9 an Armstrong number? {is_armstrong_number(9)}")  
# print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}")  
# print(f"Is 370 an Armstrong number? {is_armstrong_number(370)}")  
# print(f"Is 371 an Armstrong number? {is_armstrong_number(371)}")  
# print(f"Is 407 an Armstrong number? {is_armstrong_number(407)}")  
# print(f"Is 1634 an Armstrong number? {is_armstrong_number(1634)}")  
# print(f"Is 123 an Armstrong number? {is_armstrong_number(123)}")  
# print(f"Is -153 an Armstrong number? {is_armstrong_number(-153)}")  
# print(f"Is 0 an Armstrong number? {is_armstrong_number(0)}")
```

Output:

Enter a number to check if it's an Armstrong number: 153 153 is an Armstrong number.

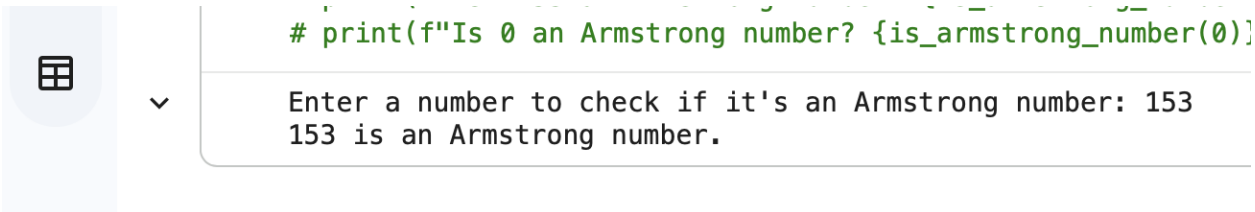
```
[4] def is_armstrong_number(number):  
    """  
    Checks if a given number is an Armstrong number.  
  
    An Armstrong number (also known as a narcissistic number) is a number  
    that is the sum of its own digits each raised to the power of the  
    number of digits.  
  
    Args:  
        number (int): The number to check.  
  
    Returns:  
        bool: True if the number is an Armstrong number, False otherwise.  
    """  
    if not isinstance(number, int) or number < 0:  
        return False # Armstrong numbers are typically positive integers  
  
    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  
  
# Example usage:  
print(f"Is 9 an Armstrong number? {is_armstrong_number(9)}") # 9^1 = 9  
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}") # 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153  
print(f"Is 370 an Armstrong number? {is_armstrong_number(370)}") # 3^3 + 7^3 + 0^3 = 27 + 343 + 0 = 370  
print(f"Is 371 an Armstrong number? {is_armstrong_number(371)}") # 3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371  
print(f"Is 407 an Armstrong number? {is_armstrong_number(407)}") # 4^3 + 0^3 + 7^3 = 64 + 0 + 343 = 407  
print(f"Is 1634 an Armstrong number? {is_armstrong_number(1634)}") # 1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634  
print(f"Is 123 an Armstrong number? {is_armstrong_number(123)}") # Not an Armstrong number  
print(f"Is -153 an Armstrong number? {is_armstrong_number(-153)}")  
print(f"Is 0 an Armstrong number? {is_armstrong_number(0)}")
```

```
... Is 9 an Armstrong number? True  
Is 153 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 -153 an Armstrong number? False  
Is 0 an Armstrong number? True
```



The image shows a Visual Studio Code editor window with a Python script. The script defines a function `is_armstrong_number` that checks if a given number is an Armstrong number. It includes docstrings, type hints, and a main block that takes user input and prints the result. The script is executed, and the output is shown in the console.

```
[7] ✓ 5s ▶ def is_armstrong_number(number):  
    """  
    Checks if a given number is an Armstrong number (another version).  
  
    An Armstrong number (also known as a 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 number to check.  
  
    Returns:  
        bool: True if the number is an Armstrong number, False otherwise.  
    """  
    if not isinstance(number, int) or number < 0:  
        return False  
  
    original_number = number  
    num_digits = len(str(number))  
    sum_of_powers = 0  
  
    while number > 0:  
        digit = number % 10  
        sum_of_powers += digit ** num_digits  
        number //= 10  
  
    return sum_of_powers == original_number  
  
# Get input from the user  
try:  
    user_input = int(input("Enter a number to check if it's an Armstrong number: "))  
    if is_armstrong_number(user_input):  
        print(f"{user_input} is an Armstrong number.")  
    else:  
        print(f"{user_input} is not an Armstrong number.")  
except ValueError:  
    print("Invalid input. Please enter an integer.")
```



The terminal output shows the execution of the script. It prompts the user to enter a number, and the user enters 153. The script then prints "153 is an Armstrong number.".

```
# print(f"Is 0 an Armstrong number? {is_armstrong_number(0)}")  
  
Enter a number to check if it's an Armstrong number: 153  
153 is an Armstrong number.
```

Explanation:

Both Gemini and GitHub Copilot correctly generated Armstrong number checker programs.

Gemini's code is easy to understand with step-by-step logic, making it suitable for beginners.

GitHub Copilot's code is shorter and more optimized but slightly harder to understand.

Overall, Gemini focuses on clarity while Copilot focuses on efficiency.

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

```
#write a python program to check if a year is leap year
#creating a python program to check if a year is a 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.
        A leap year is:
        - Divisible by 4 AND not divisible by 100
        - OR divisible by 400
    """
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    else:
        return False

# Get user input for the year
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.")

# Example usage with additional test cases
test_years = [2000, 1900, 2020, 2024, 2023]
print("\nTest cases:")
for y in test_years:
    result = "is" if is_leap_year(y) else "is not"
    print(f"{y} {result} a leap year.")
```

Output:

```
Enter a year to check if it's a leap year: 2025
2025 is not a leap year.
```


A2(1).py > ...

```
1  # Write a python program to check if a year is leap year
2
3  def is_leap_year(year):
4      """
5      Check if a given year is a leap year.
6
7      Parameters:
8      year (int): The year to check.
9
10     Returns:
11     bool: True if the year is a leap year, False otherwise.
12
13     A leap year is:
14     - Divisible by 4 AND not divisible by 100
15     - OR divisible by 400
16     """
17     if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
18         return True
19     else:
20         return False
21
22 # Get user input for the year
23 year = int(input("Enter a year to check if it's a leap year: "))
24 if is_leap_year(year):
25     print(f"{year} is a leap year.")
26 else:
27     print(f"{year} is not a leap year.")
28
29 # Example usage with additional test cases
30 test_years = [2000, 1900, 2020, 2024, 2023]
31 print("\nTest cases:")
32 for y in test_years:
33     result = "is" if is_leap_year(y) else "is not"
34     print(f"{y} {result} a leap year.")
35
```

Open file in editor (cmd + click)

INSOLE

TERMINAL

PORTS

GITLENS

/usr/bin/python3 "/Users/saivenkatesh/Documents/AI Coding/LAB Assignments/A2(1).py"

saivenkatesh@Saivs-MacBook-Air LAB Assignments % /usr/bin/python3 "/Users/saivenkatesh/Documents/AI Coding/LAB Assignments/A2(1).py"

Enter a year to check if it's a leap year: 2025

2025 is not a leap year.

--- Test Cases ---

2000 is a leap year.

1900 is not a leap year.

2020 is a leap year.

2024 is a leap year.

2023 is not a leap year.

2004 is a leap year.

1800 is not a leap year.

Explanation:

Cursor AI generated correct leap year validation code for both prompts.

With a simple prompt, the code was basic and direct, while the detailed prompt produced cleaner and more reusable function-based code.

This shows that Cursor AI changes its coding style based on how the prompt is written.

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

Prompt:

```
#write a python program that calculates the sum of odd and even numbers in tuple
```

Code:

```
def sum_odd_even (numbers) :  
    """Calculate the sum of odd and even numbers in a tuple"""  
    sum_odd = 0  
    sum_even = 0  
    for num in numbers:  
        if num % 2 == 0:  
            sum_even += num  
        else:  
            sum_odd += num  
    return sum_odd, sum_even  
# Example tuple  
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
sum_odd, sum_even = sum_odd_even (numbers)  
print(f"Sum of odd numbers: {sum_odd} ")  
print (f"Sum of even numbers: {sum_even} ")
```

Output:

Sum of odd numbers: 25

Sum of even numbers: 30

```
#Write a Python program that calculates the sum of odd and even numbers in a tuple
def sum_odd_even(numbers):
    """Calculate the sum of odd and even numbers in a tuple"""
    sum_odd = 0
    sum_even = 0

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

    return sum_odd, sum_even

# Example tuple
numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
sum_odd, sum_even = sum_odd_even(numbers)
print(f"Sum of odd numbers: {sum_odd}")
print(f"Sum of even numbers: {sum_even}")
```

```
Sum of odd numbers: 25
Sum of even numbers: 30
```

Explanation:

The original code worked correctly but was lengthy.

After AI refactoring, the code became cleaner and more efficient.

AI improved readability without changing the logic

Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.