

AI Assisted Coding

Assignment - 02

Name : G.Tejomayi

Roll No : 2303A54064

Batch : 47b

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

Task 1: Statistical Summary for Survey Data

- ❖ **Scenario:** You are a data analyst intern working with survey responses stored as numerical lists.

- **Prompt used :**

"Write a Python function that takes a list of numbers and returns the mean, minimum, and maximum values."

- **Generated code :**

```
import statistics
def analyze_numbers(numbers):
    """
    Calculates the mean, minimum, and maximum of 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

```
data = [10, 20, 30, 40, 50]
mean, minimum, maximum = analyze_numbers(data)
print(f>List: {data}")
print(f"Mean: {mean}")
print(f"Minimum: {minimum}")
print(f"Maximum: {maximum}")
```

```

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

```

- **Sample Output :**

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

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

- **Short Explanation of Logic :**

The **mean** represents the average of a set of numbers. It is obtained by adding all the values in the list and dividing the sum by the total number of elements.

The **minimum value** is the smallest number present in the list, and the **maximum value** is the largest number present in the list. These values are calculated using Python's built-in `min()` and `max()` functions.

The function accepts a list of numerical values as input, computes the mean, minimum, and maximum values, and displays them as a statistical summary of the given data.

The screenshot shows a Jupyter Notebook interface in Google Colab. The code cell contains the following Python function:

```

import statistics

def analyze_numbers(numbers):
    """
    Calculates the mean, minimum, and maximum of 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

```

The output cell shows the execution of the function with a sample list [10, 20, 30, 40, 50]. The output is:

```

[1]: data = [10, 20, 30, 40, 50]
      mean, minimum, maximum = analyze_numbers(data)

      print("List: ", data)
      print("Mean: ", mean)
      print("Minimum: ", minimum)
      print("Maximum: ", maximum)

      data_empty = []
      mean_empty, min_empty, max_empty = analyze_numbers(data_empty)
      print("List: ", data_empty)
      print("Mean: ", mean_empty)
      print("Minimum: ", min_empty)
      print("Maximum: ", max_empty)

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

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

```

A Gemini 2.5 AI sidebar is visible on the right, providing a summary of the function and asking for help with building it.

Task 2: Armstrong Number – AI Comparison

- ❖ **Scenario : You are evaluating AI tools for numeric validation logic.**

- **Prompt used (Gemini):**

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

- **Prompt used (Copilot):**

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

- **Generated code (Gemini):**

```
def is_armstrong_number(number):
    """
```

Checks if a given number is an Armstrong number.

An Armstrong number (also known as a narcissistic number, pluperfect digital invariant, or plus perfect 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
```

```
    num_str = str(number)
```

```
    n = len(num_str)
```

```
    sum_of_powers = 0
```

```
    for digit_char in num_str:
```

```
        digit = int(digit_char)
```

```
        sum_of_powers += digit ** n
```

```
    return sum_of_powers == number
```

- **Generated code (Copilot):**

```
#!/usr/bin/env python3
"""Armstrong number checker (task2).
```

Usage:

```
python3 task2.py 153
```

```
python3 task2.py # prompts for input
```

```
"""
```

```
import argparse
```

```
import sys
```

```
def is_armstrong(n: int) -> bool:
    if n < 0:
        return False
    digits = [int(d) for d in str(n)]
    p = len(digits)
    return sum(d ** p for d in digits) == n
```

```
def main() -> None:
```

```
    parser = argparse.ArgumentParser(description='Check Armstrong numbers')
```

```
    parser.add_argument('number', nargs='?', type=int, help='integer to check')
```

```
    args = parser.parse_args()
```

```

if args.number is None:
    try:
        s = input('Enter a non-negative integer: ').strip()
        num = int(s)
    except (EOFError, ValueError):
        print('Invalid input', file=sys.stderr)
        sys.exit(1)
    else:
        num = args.number

    if is_armstrong(num):
        print(f'{num} is an Armstrong number.')
    else:
        print(f'{num} is not an Armstrong number.')

if __name__ == '__main__':
    main()

```

- **Sample Input (Gemini):**

```

# Test with some numbers
print(f'Is 153 an Armstrong number? {is_armstrong_number(153)}') # Expected: True
(1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153)
print(f'Is 9474 an Armstrong number? {is_armstrong_number(9474)}') # Expected:
True (9^4 + 4^4 + 7^4 + 4^4 = 6561 + 256 + 2401 + 256 = 9474)
print(f'Is 9 an Armstrong number? {is_armstrong_number(9)}') # Expected: True (9^1
= 9)
print(f'Is 10 an Armstrong number? {is_armstrong_number(10)}') # Expected: False
(1^2 + 0^2 = 1 != 10)
print(f'Is 123 an Armstrong number? {is_armstrong_number(123)}') # Expected:
False

```

- **Sample Input (Copilot):**

Enter a non-negative integer: 343

- **Sample Output (Gemini):**

```

Is 153 an Armstrong number? True
Is 9474 an Armstrong number? True
Is 9 an Armstrong number? True
Is 10 an Armstrong number? False
Is 123 an Armstrong number? False

```

- **Sample Output (Copilot):**

343 is not an Armstrong number.

- **Side-by-Side Comparison Table :**

Feature	Google Gemini	GitHub Copilot
Code structure	Uses a separate function	Written directly in main logic
Readability	Very clear and modular	Simple and straightforward
Logic clarity	Easy to understand with function	Easy but less modular
Beginner friendly	High	Medium
Explanation support	Provides explanation along with code	No explanation, only code
Best use case	Learning and documentation	Fast coding inside editor

- **Side-by-Side Comparison Table :**

Analysis : Google Gemini generated a well-structured and modular solution using a function, making it suitable for learning and reuse. GitHub Copilot produced a concise and direct implementation, which is efficient for quick development within an IDE. Both tools correctly implemented the Armstrong number logic, but Gemini provided better clarity and explanation, while Copilot focused on speed and simplicity.

Conclusion : Both AI tools effectively generated correct Armstrong number checking programs. Google Gemini is more helpful for understanding and learning due to its explanations, while GitHub Copilot is ideal for rapid code generation during development.

The screenshot shows a Google Colab notebook titled "AI LAB2.ipynb". On the left, the code for a function named `is_armstrong_number` is displayed. The function checks if a given number is an Armstrong number by calculating the sum of its digits each raised to the power of the number of digits. It includes docstrings and example usage at the bottom. On the right, the Gemini AI interface is shown, providing a detailed explanation of the task and the generated Python code. The GitHub Copilot interface is also visible, showing a very compact one-line implementation of the same logic.

```

def is_armstrong_number(number):
    """
    Checks if a given number is an Armstrong number.

    An Armstrong number (also known as a narcissistic number, plus perfect number, or self-perfect 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

    num_str = str(number)
    n = len(num_str)
    sum_of_powers = 0

    for digit_char in num_str:
        digit = int(digit_char)
        sum_of_powers += digit ** n

    return sum_of_powers == number

# Example Usage
# Test with some numbers
print("Is 153 an Armstrong number? ", is_armstrong_number(153)) # Expected: True (1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153)
print("Is 9474 an Armstrong number? ", is_armstrong_number(9474)) # Expected: True (9^4 + 4^4 + 7^4 + 4^4 = 6561 + 256 + ...
print("Is 9 an Armstrong number? ", is_armstrong_number(9)) # Expected: True (9^1 = 9)
print("Is 10 an Armstrong number? ", is_armstrong_number(10)) # Expected: False (1^2 + 0^2 = 1 != 10)
print("Is 123 an Armstrong number? ", is_armstrong_number(123)) # Expected: False
  
```

The screenshot shows a VS Code interface with the following details:

- EXPLORER**: Shows files `task2.py`, `task2.py > ...`, `armstrong.py`, and `LAB2.1`.
- CODE**: The `task2.py` file is open, containing code to check if a number is Armstrong. It uses `argparse` to handle command-line arguments and `sum` with a generator expression to calculate the sum of digits raised to the power of their count.
- TERMINAL**: Shows a terminal window with the following session:

```
anumandararithika@ANUMANDLAS-MacBook-Air lab2.1 % /usr/bin/python3 /Users/anumandararithika/SRUI/AI_Assisted_lab/lab2.1/task2.py
KeyboardInterrupt
anumandararithika@ANUMANDLAS-MacBook-Air lab2.1 % /usr/bin/python3 /Users/anumandararithika/SRUI/AI_Assisted_lab/lab2.1/task2.py
Enter a non-negative integer: 343
343 is not an Armstrong number.
anumandararithika@ANUMANDLAS-MacBook-Air lab2.1 %
```
- OUTPUT**: Shows the output of the terminal command: "343 is not an Armstrong number."
- DEBUG CONSOLE**: Not visible in the screenshot.
- TERMINAL**: Shows the terminal window with the session above.
- PORTS**: Not visible in the screenshot.
- CHAT**: Shows a message from a user asking to write a Python program to check Armstrong numbers, followed by a note from the developer about adding tests.
- PROBLEMS**: Shows no problems.
- OUTPUT**: Shows no output.
- DEBUG CONSOLE**: Not visible in the screenshot.
- TERMINAL**: Shows the terminal window with the session above.
- PORTS**: Not visible in the screenshot.
- PROBLEMS**: Shows no problems.
- OUTPUT**: Shows no output.
- DEBUG CONSOLE**: Not visible in the screenshot.
- TERMINAL**: Shows the terminal window with the session above.
- PORTS**: Not visible in the screenshot.

Task 3: Leap Year Validation Using Cursor API

- ❖ Scenario : You are validating a calendar module for a backend system.

- **Prompt 1:**

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

- **Generated code :**

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

A year is a leap year if:

- It is divisible by 4 AND

- not divisible by 100, UNLESS

- it is also divisible by 400

```
if(year % 4 == 0 and year % 100
```

```
print(f"\{year\} is a leap year.")
```

se:

pr

prompt 2:

- "Write an o-

Write an optimized Python program with proper conditions and comments to check whether a year is a leap year."

- ## • Generated code :

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

```
# A year is a leap year if:  
# 1. It is divisible by 400 → leap year  
# 2. Else if it is divisible by 100 → not a leap year  
# 3. Else if it is divisible by 4 → leap year  
# 4. Otherwise → not a leap year
```

```

if year % 400 == 0:
    print(f"{year} is a leap year.")
elif year % 100 == 0:
    print(f"{year} is not a leap year.")
elif year % 4 == 0:
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")

```

- **Sample Input :**

Enter a year: 2026

- **Sample Output :**

2026 is not a leap year.

- **Short Explanation of Logic :**

The first version of the code generated by Cursor AI provided basic leap year validation logic. The second prompt resulted in improved code with better readability and clear comments explaining the conditions. The optimized version is easier to understand and more suitable for real-world applications.

The screenshot shows a VS Code interface with two tabs: 'lepyear.py' and 'leap_year.py'. The 'lepyear.py' tab contains the initial AI-generated code. The 'leap_year.py' tab shows the refined code with comments and a cursor AI chat window. The terminal at the bottom shows the execution of the program with input '2026'.

```

lepyear.py
1 year = int(input("Enter a year: "))
2
3 # A year is a leap year if:
4 # - It is divisible by 4 AND
5 # - not divisible by 100, UNLESS
6 # - it is also divisible by 400
7 if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
8     print(f"{year} is a leap year.")
9 else:
10    print(f"{year} is not a leap year.")

leap_year.py
1m · Auto
"Write a Python program to check whether a year is a leap year."
Creating a simple program 3s
Python leap year program
year = int(input("Enter a year: "))

# A year is a leap year if:
# - It is divisible by 4 AND
# - not divisible by 100, UNLESS
# - it is also divisible by 400
if (year % 4 == 0 and year % 100 != 0):
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")

Terminal
anumandararithika@ANUMANDLA8-MacBook-Air leap_year.py % /usr/bin/python3 /Users/anumandararithika/SRU/AI_Assigned_lab/lab2.1/leap_year.py/lepyear.py
Enter a year: 2026
2026 is not a leap year.
anumandararithika@ANUMANDLA8-MacBook-Air leap_year.py %

```

```

leapyear.py
1 # Program to check if a given year is a leap year
2
3 year = int(input("Enter a year: "))
4
5 # A year is a leap year if:
6 # 1. It is divisible by 400           - leap year
7 # 2. Else if it is divisible by 100  - not a leap year
8 # 3. Else if it is divisible by 4   - leap year
9 # 4. Otherwise                      - not a leap year
10
11 if year % 400 == 0:
12     print(f"{year} is a leap year.")
13 elif year % 100 == 0:
14     print(f"{year} is not a leap year.")
15 elif year % 4 == 0:
16     print(f"{year} is a leap year.")
17 else:
18     print(f"{year} is not a leap year.")

# Program to check if a given year is a leap year
year = int(input("Enter a year: "))

# A year is a leap year if:
# 1. It is divisible by 400
# 2. Else if it is divisible by 100
# 3. Else if it is divisible by 4
# 4. Otherwise

if year % 400 == 0:
    print(f"{year} is a leap year.")
elif year % 100 == 0:
    print(f"{year} is not a leap year")
elif year % 4 == 0:
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year")

```

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

- ❖ Scenario : Company policy requires developers to write logic before using AI.

- Prompt used :

"Refactor this Python code to improve readability and efficiency."

- Student Code:

$T = (1, 2, 3, 4, 5, 6, 7)$

even_sum = 0

odd_sum = 0

for i in t:

if i % 2 == 0:

 even_sum = even_sum + i

else:

 odd_sum = odd_sum + i

print("Sum of even numbers:", even_sum)

print("Sum of odd numbers:", odd_sum)

- AI Code:

```
t = (1, 2, 3, 4, 5, 6, 7)

# Using generator expressions with sum for clarity and efficiency
even_sum = sum(i for i in t if i % 2 == 0)
odd_sum = sum(i for i in t if i % 2 != 0)

print("Sum of even numbers:", even_sum)
print("Sum of odd numbers:", odd_sum)
```

- **Sample Output :**

Sum of even numbers: 12
 Sum of odd numbers: 16

The screenshot shows the VS Code interface with the following details:

- Left Panel (Lépyear.py):** Displays the Python script `lepyear.py` containing the original code:


```
t = (1, 2, 3, 4, 5, 6, 7)
...
print("Sum of even numbers:", even_sum)
print("Sum of odd numbers:", odd_sum)
```
- Right Panel (Python program for leap year che...):** Shows a tooltip for the refactored code. The refactored code uses generator expressions and the `sum` function:


```
t = (1, 2, 3, 4, 5, 6, 7)
...
even_sum = sum(i for i in t if i % 2 == 0)
odd_sum = sum(i for i in t if i % 2 != 0)

print("Sum of even numbers:", even_sum)
print("Sum of odd numbers:", odd_sum)
```
- Bottom Panel (Terminal):** Shows the terminal output for the year 2026, indicating it is not a leap year. It also shows the execution of the script for years 2026 and 2024, with the output "2024 is a leap year".