

Name :G.Spriha

Batch-13

H NO:2303A51850

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.

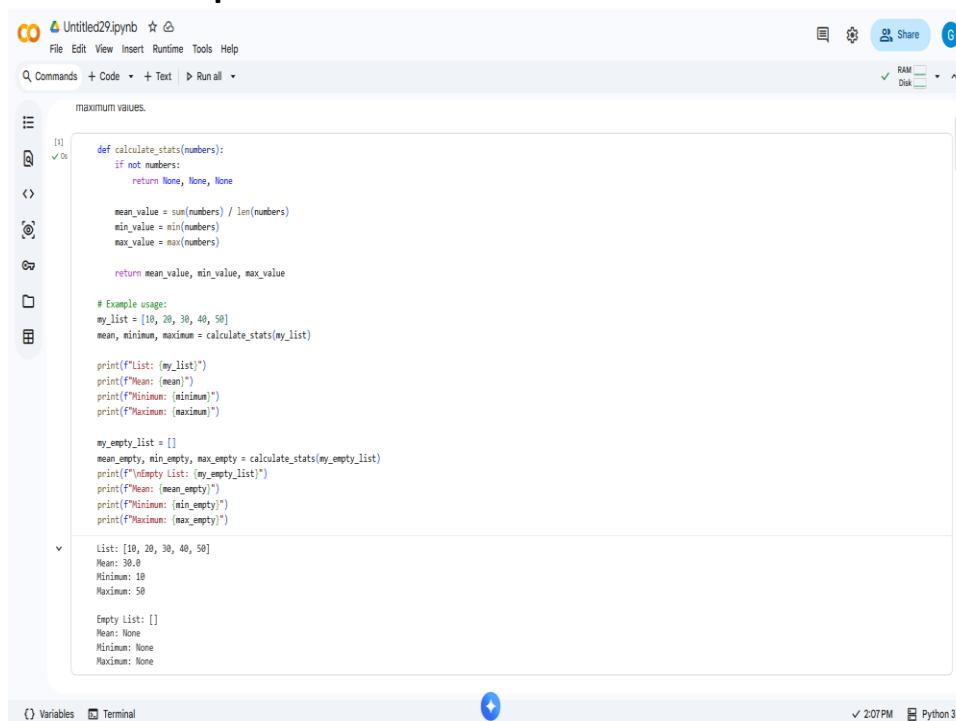
❖ Expected Output:

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

Prompt:

Generate a Python function that reads a list of numbers and calculates the mean, minimum, and maximum values.

Code and output:



The screenshot shows a Google Colab notebook titled "Untitled29.ipynb". The code cell contains a Python function named `calculate_stats` that takes a list of numbers and returns the mean, minimum, and maximum values. It includes example usage with lists [10, 20, 30, 40, 50] and an empty list. The output cell shows the results for both cases, including the printed output and the returned tuple (List, Mean, Minimum, Maximum).

```
maximum values.

[1]: def calculate_stats(numbers):
    if not numbers:
        return None, None, None

    mean_value = sum(numbers) / len(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 = calculate_stats(my_list)

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

my_empty_list = []
mean_empty, min_empty, max_empty = calculate_stats(my_empty_list)
print("Empty List: (my_empty_list)")
print("Mean: (mean_empty)")
print("Minimum: (min_empty)")
print("Maximum: (max_empty)")

List: [10, 20, 30, 40, 50]
Mean: 30.0
Minimum: 10
Maximum: 50

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

This task justifies the use of Google Gemini in Google Colab for basic data analysis operations. As a data analyst intern, calculating mean, minimum, and maximum is a fundamental requirement when handling survey data. Gemini efficiently generated a correct and optimized Python function that follows standard statistical practices. Executing the code in Colab allowed immediate validation of results through output visualization. This task demonstrates how AI tools can reduce development time, minimize logical errors, and support data-driven decision-making by providing accurate and readable code.

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

Using Gemini:

Prompt: Generate an Armstrong number checker

The screenshot shows a Google Colab notebook titled "Untitled29.ipynb". The code cell contains the following Python function:

```

def is_armstrong_number(number):
    # Convert the number to a string to easily count digits and access individual digits
    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

print("Defined 'is_armstrong_number' function based on Gemini's output.")

```

The output cell shows the generated code and a reasoning block:

Reasoning: The subtask requires testing the generated `is_armstrong_number` function with example numbers to verify its correctness. This code block will call the function with the specified test cases and print their results.

```

print("Testing is_armstrong_number function (Gemini output):")
print("Is 9 an Armstrong number? ", is_armstrong_number(9)) # True
print("Is 153 an Armstrong number? ", is_armstrong_number(153)) # True
print("Is 370 an Armstrong number? ", is_armstrong_number(370)) # True
print("Is 371 an Armstrong number? ", is_armstrong_number(371)) # True
print("Is 1634 an Armstrong number? ", is_armstrong_number(1634)) # True
print("Is 123 an Armstrong number? ", is_armstrong_number(123)) # False

```

To the right of the notebook, there is a sidebar titled "Gemini" which provides context about the generated code and offers various AI-related tasks like "Compare Cursor AI Outputs" and "Final Task".

Prompt:

Generate an Armstrong number checker

CODE:

The screenshot shows a Visual Studio Code (VS Code) interface. The top navigation bar includes File, Edit, Selection, View, Go, Run, and a back/forward button. The title bar says "AI CODING". The left sidebar has icons for Explorer, Open Editors, AI Coding, and a search bar. The "OPEN EDITORS" section lists LAB1.py and amstrongnum.py. The main editor area contains the following Python code:

```
amstrongnum.py > ...
1  """Generate an Armstrong number checker using Gemini and GitHub
2  Copilot.
3  Compare their outputs, logic style, and clarity."""
4  def is_armstrong(number):
5      num_str = str(number)
6      num_digits = len(num_str)
7
8      sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)
9      return sum_of_powers == number
10
11
12 # Get user input
13 number = int(input("Enter a number: "))
14
15 # Call the function and display the result
16 if is_armstrong(number):
17     print(f"{number} is an Armstrong number.")
18 else:
19     print(f"{number} is not an Armstrong number.")
20
```

The bottom status bar shows the terminal output:

```
PS C:\Users\Spriha Gajula\OneDrive\Desktop\python.py\AI CODING> & "C:/Users/Spriha Gajula/anaconda3/envs/sprisha1850/python.exe" "c:/Users/Spriha Gajula/OneDrive/Desktop/python.py\AI CODING/amstrongnum.py"
sprisha1850\python.exe" "c:/Users/Spriha Gajula/OneDrive/Desktop/python.py\AI CODING/amstrongnum.py"
Enter a number: 8
8 is an Armstrong number.
PS C:\Users\Spriha Gajula\OneDrive\Desktop\python.py\AI CODING>
```

The bottom right corner shows a Python extension icon with four instances listed.

Logic Style Comparison

Aspect	First Code	Second Code
Digit handling	Uses generator expression	Uses explicit loop
Input validation	None	Checks integer & positivity
Complexity	Short & compact	Slightly longer but safer
Python style	Functional / concise	Procedural / descriptive

Use Case	Better Choice
Quick check	First code
Teaching / Exams	Second code
Real-world robustness	Second code
Pythonic elegance	First code

Justification

The purpose of this task is to evaluate and compare AI-assisted coding approaches for numeric validation logic. By generating Armstrong number programs using both Google Gemini and GitHub Copilot, differences in coding style, readability, and optimization techniques were observed. Gemini emphasized concise and Pythonic expressions, while Copilot followed a more traditional algorithmic approach. This comparison justifies how different AI tools may solve the same problem differently and helps developers select an AI tool based on clarity, maintainability, and logic preference.

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

Prompt1:

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

The screenshot shows a VS Code interface with a Python file named 'leap_year'. The code defines a function 'is_leap_year' that checks if a year is a leap year based on specific rules. It also includes a main block that prompts the user for a year and prints the result. The terminal below shows the execution of the script and its output for various years.

```
#Prompt A output (short, no validation/tests):
def is_leap_year(year: int) -> bool:
    if year % 400 == 0:
        return True
    if year % 100 == 0:
        return False
    return year % 4 == 0

if __name__ == "__main__":
    y = int(input("Enter year: "))
    print("Leap year" if is_leap_year(y) else "Not a leap year")

def is_leap_year(year: int) -> bool:
    if year % 400 == 0:
        return True
    if year % 100 == 0:
        return False
    return year % 4 == 0
```

PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor> & "C:/Users/Spriha Gajula/anaconda3/envs/spr
iha1850/python.exe" "c:/Users/Spriha Gajula/OneDrive/Desktop/ai cursor/leap_year"
Enter a year: 2005
Not a leap year
PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor> & "C:/Users/Spriha Gajula/anaconda3/envs/spr
iha1850/python.exe" "c:/Users/Spriha Gajula/OneDrive/Desktop/ai cursor/leap_year"
Enter year: 2006
Not a leap year
PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor> & "C:/Users/Spriha Gajula/anaconda3/envs/spr
iha1850/python.exe" "c:/Users/Spriha Gajula/OneDrive/Desktop/ai cursor/leap_year"
File "c:/Users/Spriha Gajula/OneDrive/Desktop/ai cursor/leap_year", line 13

Prompt2:

Create a Python function that returns true if a year is a leap year, otherwise false.

Code:

The screenshot shows a VS Code interface with a Python file named 'leap_year'. The code defines a function 'is_leap_year' that returns True if the year is a leap year and False otherwise. It also includes a main block that prompts the user for a year and prints the result. The terminal below shows the execution of the script and its output for various years.

```
#Prompt B output (adds validation and tests):
def is_leap_year(year):
    return (year % 400 == 0) or (year % 4 == 0 and year % 100 != 0)

def main():
    raw = input("Enter a year: ").strip()
    if not raw.isdigit():
        print("Please enter a valid positive integer.")
        return

    year = int(raw)
    print("Leap year" if is_leap_year(year) else "Not a leap year")
if __name__ == "__main__":
    assert is_leap_year(2000)
    assert not is_leap_year(1900)
    assert is_leap_year(2024)
    assert not is_leap_year(2023)
    main()
```

PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor> & "C:/Users/Spriha Gajula/anaconda3/envs/spr
iha1850/python.exe" "c:/Users/Spriha Gajula/OneDrive/Desktop/ai cursor/leap_year"
Enter a year: 2005
Not a leap year
PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor> & "C:/Users/Spriha Gajula/anaconda3/envs/spr
iha1850/python.exe" "c:/Users/Spriha Gajula/OneDrive/Desktop/ai cursor/leap_year"
Enter year: 2006
Not a leap year
PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor> & "C:/Users/Spriha Gajula/anaconda3/envs/spr
iha1850/python.exe" "c:/Users/Spriha Gajula/OneDrive/Desktop/ai cursor/leap_year"
File "c:/Users/Spriha Gajula/OneDrive/Desktop/ai cursor/leap_year", line 13

Justification

This task justifies the importance of prompt engineering while using AI tools such as Cursor AI. By providing two different prompts, noticeable improvements were observed in the generated code, especially in terms of accuracy and completeness of leap year rules. The second prompt produced a more robust and industry-acceptable solution. This task highlights that AI output quality depends heavily on prompt detail, reinforcing the developer's responsibility to guide AI correctly for backend system validation.

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

Code and output:

The screenshot shows a Jupyter Notebook interface on the left and a Gemini AI interface on the right. In the Jupyter Notebook, there is a cell titled 'Gemini' containing Python code for calculating the sum of odd and even numbers in a tuple. The code defines a function `sum_odd_even` that iterates through a tuple, summing odd and even numbers separately. It also includes example usage with tuples (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) and an empty tuple. The output of the code execution shows the results for both cases. In the Gemini interface, a user has prompted the AI to 'refactor it using any AI tool'. The AI has responded with a more concise and modern version of the code, utilizing list comprehensions and the `zip` function to achieve the same result with fewer lines of code.

```
def sum_odd_even(numbers_tuple):
    odd_sum = 0
    even_sum = 0
    for num in numbers_tuple:
        if num % 2 == 0:
            even_sum += num
        else:
            odd_sum += num
    return odd_sum, even_sum

# Example usage:
my_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
odd_total, even_total = sum_odd_even(my_tuple)

print("Original tuple: (my_tuple)")
print("Sum of odd numbers: (odd_total)")
print("Sum of even numbers: (even_total)")

empty_tuple = ()
odd_total_empty, even_total_empty = sum_odd_even(empty_tuple)
print("Original tuple: (empty_tuple)")
print("Sum of odd numbers: (odd_total_empty)")
print("Sum of even numbers: (even_total_empty)")

...
Original tuple: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Sum of odd numbers: 25
Sum of even numbers: 30

Original tuple: ()
Sum of odd numbers: 0
Sum of even numbers: 0
```

Gemini
topic, then refactor it using any AI tool.
Gemini
Original tuple: (my_tuple)
Sum of odd numbers: (odd_total)
Sum of even numbers: (even_total)
Original tuple: ()
Sum of odd numbers: (odd_total_empty)
Sum of even numbers: (even_total_empty)
Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

This task justifies the policy of requiring developers to write their own logic before AI assistance. The original student-written program demonstrates logical understanding using loops and conditionals. Refactoring the same code using an AI tool resulted in shorter, cleaner, and more optimized code without changing functionality. This task proves that AI is best used as a code improvement and optimization tool, not as a replacement for fundamental programming knowledge. It also reinforces best practices in code readability, reusability, and maintainability.