AI Assistant Coding

Assignment 2.1

Name : G. Hari Priya **HT. No :** 2303A51104 **Batch:** 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:

```
write a python script that reads list of numbers and caluclate mean minimum and maximum
```

Code:

```python
import statistics

# Example list of numbers
numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# Calculate the mean
mean_value = statistics.mean(numbers)

# Calculate the minimum value
min_value = min(numbers)

# Calculate the maximum value
max_value = max(numbers)

# Print the results
print(f"The list of numbers is: {numbers}")
print(f"Mean: {mean_value}")
print(f"Minimum: {min_value}")
print(f"Maximum: {max_value}")
```

Output:

```
The list of numbers is: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Mean: 55
Minimum: 10
Maximum: 100
```

Task 2: Armstrong Number — Al Comparison

Scenario:

You are evaluating Al tools for numeric validation logic.

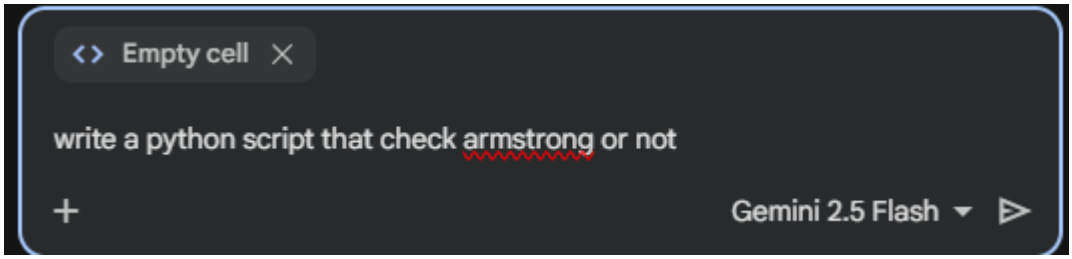
Task:

Generate an Armstrong number checker using Gemini and GitHub

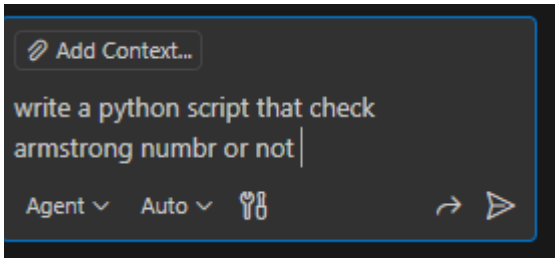
Copilot.

Compare their outputs, logic style, and clarity.

Prompt: Gemini



Copilot :



Code: Gemini

```python
def is_armstrong(number):
    # Convert number to string to find the number of digits
    num_str = str(number)
    n = len(num_str)

    sum_of_powers = 0
    temp_num = number

    while temp_num > 0:
        digit = temp_num % 10
        sum_of_powers += digit ** n
        temp_num //= 10

    return number == sum_of_powers

# Get input from the user
num = int(input("Enter a number: "))

# Check if it's an Armstrong number
if is_armstrong(num):
    print(f"{num} is an Armstrong number.")
else:
    print(f"{num} is not an Armstrong number.")
```

Copilot:

armstrong.py

armstrong.py > is_armstrong

```python
def is_armstrong(number):
    """
    Check if a number is an Armstrong number.
    An Armstrong number is equal to the sum of its digits raised to the power of the number of digits.
    Example: 153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153
    """
    # Convert to string to get individual digits
    num_str = str(number)
    num_digits = len(num_str)

    # Calculate sum of each digit raised to the power of number of digits
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)

    # Check if sum equals the original number
    return sum_of_powers == number

# Main program
if __name__ == "__main__":
    # Get input from user
    num = int(input("Enter a number to check if it's an Armstrong number: "))

    if is_armstrong(num):
        print(f"{num} is an Armstrong number!")

        # Show the calculation
        num_str = str(num)
        num_digits = len(num_str)
        calculation = " + ".join([f"{digit}^{num_digits}" for digit in num_str])
        result = sum(int(digit) ** num_digits for digit in num_str)
        print(f"Verification: {calculation} = {result}")
    else:
        print(f"{num} is NOT an Armstrong number.")

    # Display some examples
    print("\nExamples of Armstrong numbers:")
    print("1-digit: 1, 2, 3, 4, 5, 6, 7, 8, 9")
    print("3-digit: 153, 370, 371, 407")
    print("4-digit: 1634, 8208, 9474")
```

Output: Gemini

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

Copilot:

```
PROBLEMS  OUTPUT  PORTS  DEBUG CONSOLE  TERMINAL
PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/Activate.ps1
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe d:/Course/AIAC/armstrong.py
Enter a number to check if it's an Armstrong number: 153
153 is an Armstrong number!
Verification: 1^3 + 5^3 + 3^3 = 153

Examples of Armstrong numbers:
1-digit: 1, 2, 3, 4, 5, 6, 7, 8, 9
3-digit: 153, 370, 371, 407
4-digit: 1634, 8208, 9474
(AIAC_env) PS D:\Course\AIAC>
```

Explanation:

Feature	Google Gemini	GitHub Copilot
Logic Style	Functional & Pythonic	Procedural & loop-based
Digit Handling	Converts number to string	Uses mathematical digit extraction
Code Length	Short and concise	Slightly longer
Readability	Very high	Moderate
Python Features	Uses sum() and generator expression	Uses while loop
Beginner Friendly	Easy to understand	Slightly Harder
Efficiency	Efficient for small inputs	Efficient, more algorithmic
Clarity	Clean and modern	Traditional logic

Task 3: Leap Year Validation Using Cursor Al

Scenario:

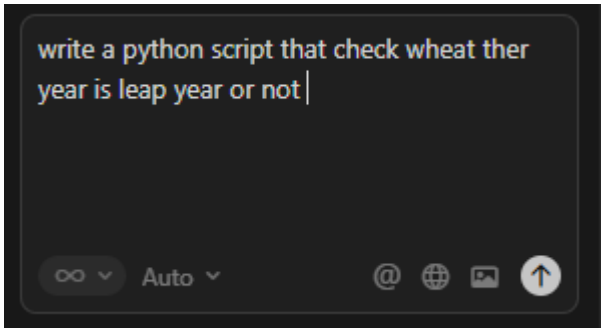
You are validating a calendar module for a backend system.

Task:

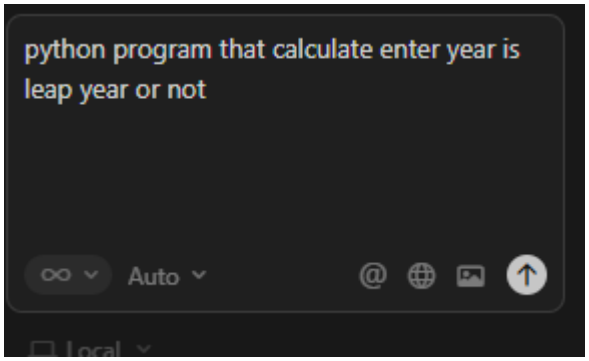
Use Cursor Al to generate a Python program that checks whether a given year is a leap year.

Use at least two different prompts and observe changes in code**.**

Prompt:**1**



2



Code:1

create_and_display_file.py > is_leap_year

```python
def is_leap_year(year):
    """
    Check if a year is a leap year.

    A year is a leap year if:
    - It is divisible by 400, OR
    - It is divisible by 4 but NOT divisible by 100

    Args:
        year (int): The year to check

    Returns:
        bool: True if the year is a leap year, False otherwise
    """
    if year % 400 == 0:
        return True
    elif year % 100 == 0:
        return False
    elif year % 4 == 0:
        return True
    else:
        return False

def main():
    """Main function to get user input and check if year is leap year."""
    try:
        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.")

    except ValueError:
        print("Invalid input! Please enter a valid year (integer).")


if __name__ == "__main__":
    main()

```

2

leap_year.py

```python
"""
Program to check if a given year is a leap year or not.

Leap year rules:
- A year is a leap year if it's divisible by 4
- However, if it's divisible by 100, it must also be divisible by 400 to be a 
- Examples: 2000 is a leap year, 1900 is not, 2024 is a leap year
"""

def is_leap_year(year):
    """
    Check if a year is a leap year.

    Args:
        year (int): The year to check

    Returns:
        bool: True if the year is a leap year, False otherwise
    """
    if year % 400 == 0:
        return True
    elif year % 100 == 0:
        return False
    elif year % 4 == 0:
        return True
    else:
        return False
def main():
    """Main function to get user input and display result."""
    try:
        # Get year input from user
        year = int(input("Enter a year: "))

        # 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.")

    except ValueError:
        print("Invalid input! Please enter a valid year (integer).")
    except Exception as e:
        print(f"An error occurred: {e}")


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

Output:1

```
Problems  Output  Debug Console  Terminal  Ports
PS D:\Course\AIAC> & D:\Course\AIAC\AIAC_env\Scripts\Activate.ps1
(AIAC_env) PS D:\Course\AIAC> & D:\Course\AIAC\AIAC_env\Scripts\python.exe d:/Course/AIAC/create_and_display_file.py
Enter a year to check if it's a leap year: 2019
2019 is not a leap year.
(AIAC_env) PS D:\Course\AIAC>
```

2

```
(AIAC_env) PS D:\Course\AIAC> & D:\Course\AIAC\AIAC_env\Scripts\python.exe d:/Course/AIAC/leap_year.py
Enter a year: 2024
2024 is a leap year.
(AIAC_env) PS D:\Course\AIAC>
```

Explanation:

There is no difference in codes

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

Scenario:

Company policy requires developers to write logic before using Al.

Task:

Write a Python program that calculates the sum of odd and even numbers in a tuple, then refactor it using any Al tool.

Code: Before refactoring

sumOfOdd.py > ...

```python
def sumOfOdd(nums):
    sumeven =0
    sumodd =0
    for num in nums:
        if num % 2 == 0:
            sumeven += num
        else:
            sumodd += num
    return sumodd, sumeven

nums = (7, 12, 19, 24, 35, 40)
sumodd, sumeven = sumOfOdd(nums)
print(f"Sum of odd numbers: {sumodd}")
print(f"Sum of even numbers: {sumeven}")
```

After Refactoring:

```python
def sum_odd_even(nums):
    """Calculate sum of odd and even numbers.

    Args:
        nums: Iterable of numbers

    Returns:
        Tuple of (sum_odd, sum_even)
    """
    sum_odd = sum(num for num in nums if num % 2 != 0)
    sum_even = sum(num for num in nums if num % 2 == 0)
    return sum_odd, sum_even


def main():
    """Main entry point."""
    nums = (7, 12, 19, 24, 35, 40)
    sum_odd, sum_even = sum_odd_even(nums)
    print(f"Sum of odd numbers: {sum_odd}")
    print(f"Sum of even numbers: {sum_even}")


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

Output: Before refactoring

```
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe d:/Course/AIAC/sumOfOdd.py
Sum of odd numbers: 61
Sum of even numbers: 76
(AIAC_env) PS D:\Course\AIAC>
```

After Refactoring:

```
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe d:/Course/AIAC/sumOfOdd.py
Sum of odd numbers: 61
Sum of even numbers: 76
(AIAC_env) PS D:\Course\AIAC>
```

Explanation:

Easy to read and understand, calculate more faster better structure