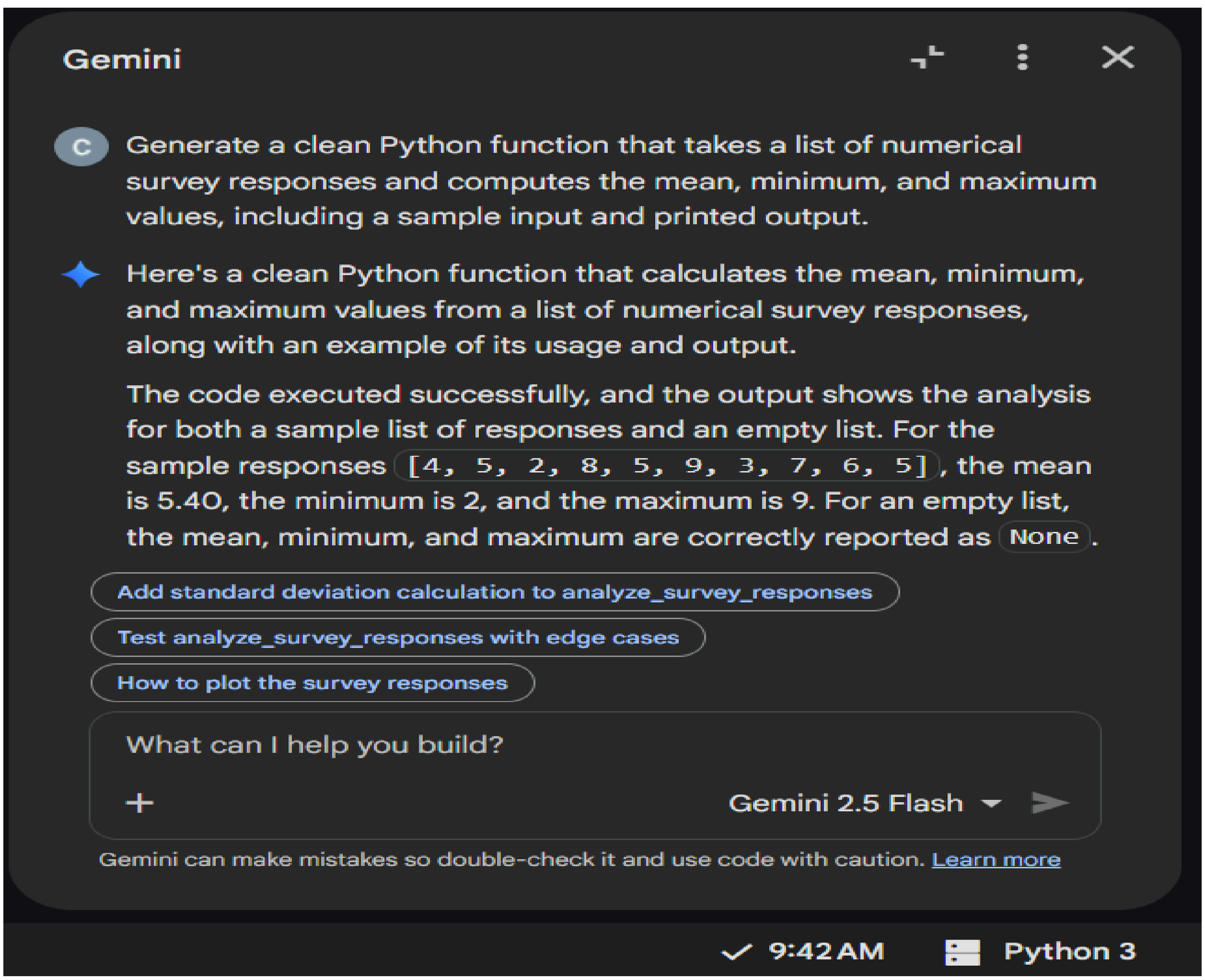
AI ASSISTED CODING
ASSIGNMENT - 2.1
ROLL NO. 2303A510J0

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

Task 1: Statistical Summary for Survey Data
Prompt:



Code:

```python
import statistics

def analyze_survey_responses(responses):
    """
    Computes the mean, minimum, and maximum of a list of numerical survey
responses.

    Args:
        responses (list): A list of numerical survey responses.

    Returns:
        dict: A dictionary containing the mean, minimum, and maximum
values.
              Returns None for min/max if the list is empty, and handles
mean accordingly.
    """
    if not responses:
        return {"mean": None, "min": None, "max": None}

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

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

# Sample Input
sample_responses = [4, 5, 2, 8, 5, 9, 3, 7, 6, 5]

# Compute and Print Output
analysis_results = analyze_survey_responses(sample_responses)
print("Survey Analysis Results:")
print(f"  Responses: {sample_responses}")
print(f"  Mean: {analysis_results['mean']:.2f}")
print(f"  Minimum: {analysis_results['min']}")
print(f"  Maximum: {analysis_results['max']}")
```

Output:

```
Survey Analysis Results:
  Responses: [4, 5, 2, 8, 5, 9, 3, 7, 6, 5]
  Mean: 5.40
  Minimum: 2
  Maximum: 9
```

Explanation:

Google Gemini generated a modular Python function using built-in functions such as `sum()`, `min()`, and `max()`.
The logic calculates the mean by dividing the total sum of elements by the number of elements.

The function-based approach improves reusability and clarity.

Task 2: Armstrong Number – AI Comparison

Github copilot:

Prompt:

```
# Write a Python function to check if a number is an Armstrong number.
# Use an efficient approach and keep the code concise.
```

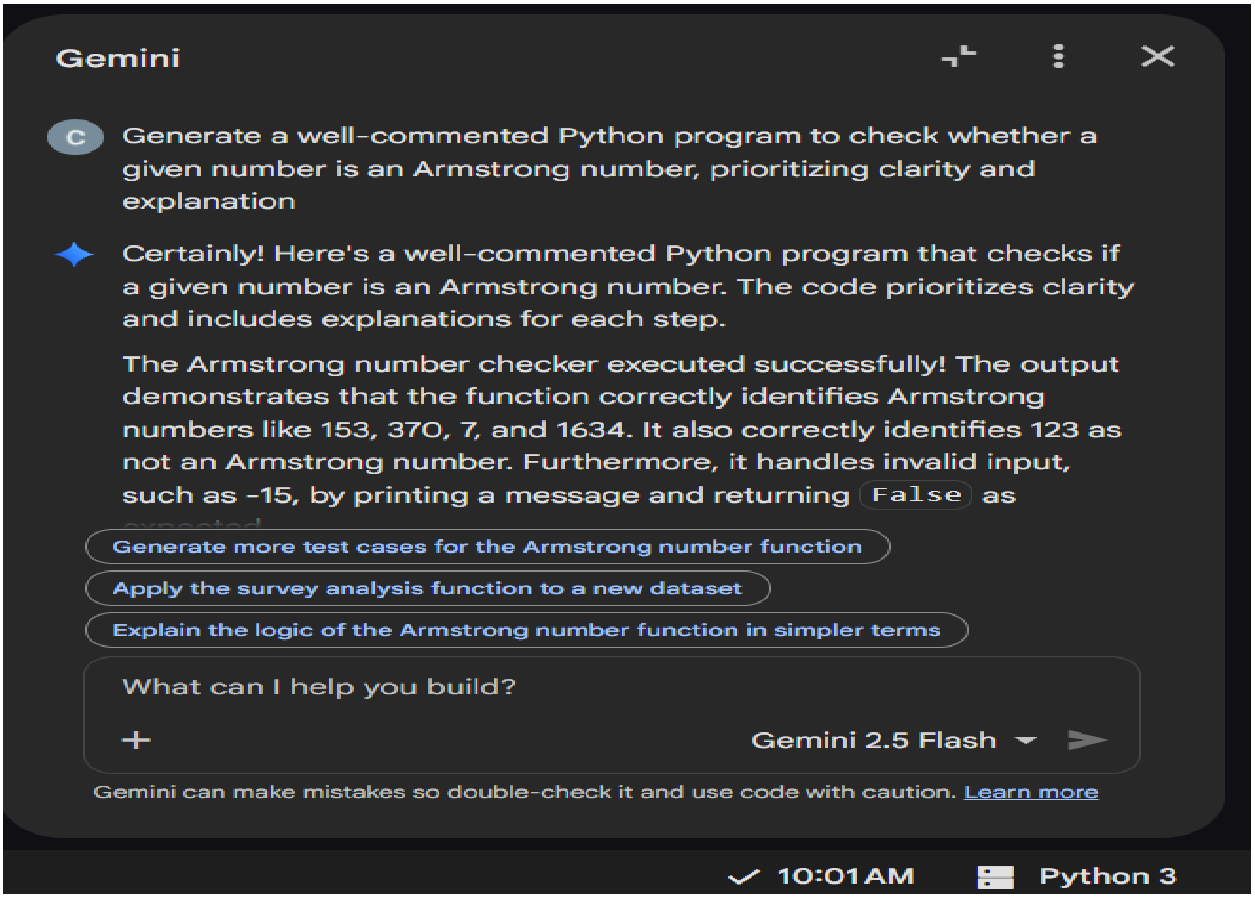
Code:

```python
# Write a Python function to check if a number is an Armstrong number.
# Use an efficient approach and keep the code concise.
def is_armstrong_number(num):
    # Convert the number to string to easily iterate over digits
    digits = str(num)
    num_digits = len(digits)

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

    # Check if the calculated sum is equal to the original number
    return armstrong_sum == num
# Example usage:
print(is_armstrong_number(153))  # True
print(is_armstrong_number(123))  # False
print(is_armstrong_number(9474)) # True
```

Output:

```
PS
C:\Users\rohit\OneDrive\Documents\SRU\ai_code\
lab_2.1> &
C:/Users/rohit/AppData/Local/Programs/Python/P
ython314/python.exe
c:/Users/rohit/OneDrive/Documents/SRU/ai_code/
lab_2.1/assignment_2_git_copilot.py
True
False
True
```

Gemini:



Code:

```
# Function to check if a number is an Armstrong number
def is_armstrong_number(number):
    """
    Checks if a given positive integer 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.

    For example, 153 is an Armstrong number because 1^3 + 5^3 + 3^3 = 1 +
125 + 27 = 153.
```

```
    Args:
        number (int): The integer to be checked.

    Returns:
        bool: True if the number is an Armstrong number, False otherwise.
    """
    # Ensure the input is a positive integer
    if not isinstance(number, int) or number < 0:
        print("Please enter a positive integer.")
        return False

    # Convert the number to a string to easily get its digits and count
them
    num_str = str(number)
    num_digits = len(num_str)

    # Initialize the sum of powers of digits
    sum_of_powers = 0

    # Iterate through each digit of the number
    for digit_char in num_str:
        # Convert the character digit back to an integer
        digit = int(digit_char)
        # Add the digit raised to the power of the number of digits to the
sum
        sum_of_powers += digit ** num_digits

    # Compare the original number with the calculated sum
    # If they are equal, it's an Armstrong number
    return number == sum_of_powers

# --- Sample Usage and Output ---

# Test with an Armstrong number
num1 = 153
print(f"Is {num1} an Armstrong number? {is_armstrong_number(num1)}")

# Test with another Armstrong number
num2 = 370
```

```python
print(f"Is {num2} an Armstrong number? {is_armstrong_number(num2)}")

# Test with a non-Armstrong number
num3 = 123
print(f"Is {num3} an Armstrong number? {is_armstrong_number(num3)}")

# Test with a single-digit number (all are Armstrong numbers)
num4 = 7
print(f"Is {num4} an Armstrong number? {is_armstrong_number(num4)}")

# Test with a larger Armstrong number
num5 = 1634
print(f"Is {num5} an Armstrong number? {is_armstrong_number(num5)}")

# Test with an invalid input
num6 = -15
print(f"Is {num6} an Armstrong number? {is_armstrong_number(num6)}")
```

Output:

```
[3]
✓ 0s
# Test with an invalid input
num6 = -15
print(f"Is {num6} an Armstrong number? {is_armstrong_number(num6)}")

Is 153 an Armstrong number? True
Is 370 an Armstrong number? True
Is 123 an Armstrong number? False
Is 7 an Armstrong number? True
Is 1634 an Armstrong number? True
Please enter a positive integer.
Is -15 an Armstrong number? False
```

Comparison

Criteria	GitHub Copilot	Google Gemini
Output	Produces correct Boolean outputs (`True` / `False`) for valid Armstrong and non-Armstrong numbers. Example outputs are	Produces correct Boolean outputs and also prints descriptive messages using formatted strings, making

	directly printed for test cases.	**outputs more user-friendly and informative.**
Logic Style	**Uses a compact and efficient approach with generator expressions and built-in `sum()` for digit power calculation. Focuses on brevity and performance.**	**Uses a step-by-step procedural approach with explicit loops and intermediate variables, emphasizing understandability and correctness.**
Clarity & Documentation	**Minimal comments, intended for developers who already understand the logic. Code is concise but less explanatory.**	**Highly documented with docstrings, inline comments, examples, and input validation, making it suitable for beginners and academic purposes.**

Conclusion:

Both GitHub Copilot and Google Gemini generate logically correct Armstrong number programs.
Copilot prioritizes concise and efficient code suitable for experienced developers, while Gemini focuses on clarity, documentation, and educational value.

Task 3: Leap Year Validation Using Cursor AI

Prompt:
Generate a minimal Python function to determine whether a given year is a leap year using standard rules, and include a simple function call to verify the logic with a sample test case.

Python leap year determination

1m · +82 · Auto

Generate a minimal Python function to determine whether a given year is a leap year using standard rules, and include a simple function call to verify the logic with a sample test case.

Auto

Checking the current file, then adding a minimal leap year function with a test case.

Read assignment_2_cursor_ai.py

Creating a minimal leap year function with a test case:

assignment_2_cursor_ai.py +19 -1

```python
def is_leap_year(year):
    """
    Determine if a given year is a leap y
```

Verifying the function works with the test case:

Ran command: cd, python

```
$ cd
  "C:\Users\rohit\OneDrive\Docum
  ents\SRU\ai_code\lab_2.1" &&
  python assignment_2_cursor_ai.
```

1 File Undo Keep Review

Plan, @ for context, / for commands

Auto

Code:

```python
def is_leap_year(year):
    """
    Determine if a given year is a leap year using standard rules.

    Rules:
    - Divisible by 400: leap year
    - Divisible by 100 but not 400: NOT a leap year
    - Divisible by 4 but not 100: leap year
    - Otherwise: NOT a leap year
    """
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)


# Test case
if __name__ == "__main__":
    test_year = 2024
    result = is_leap_year(test_year)
    print(f"{test_year} is a leap year: {result}")
```

Output:

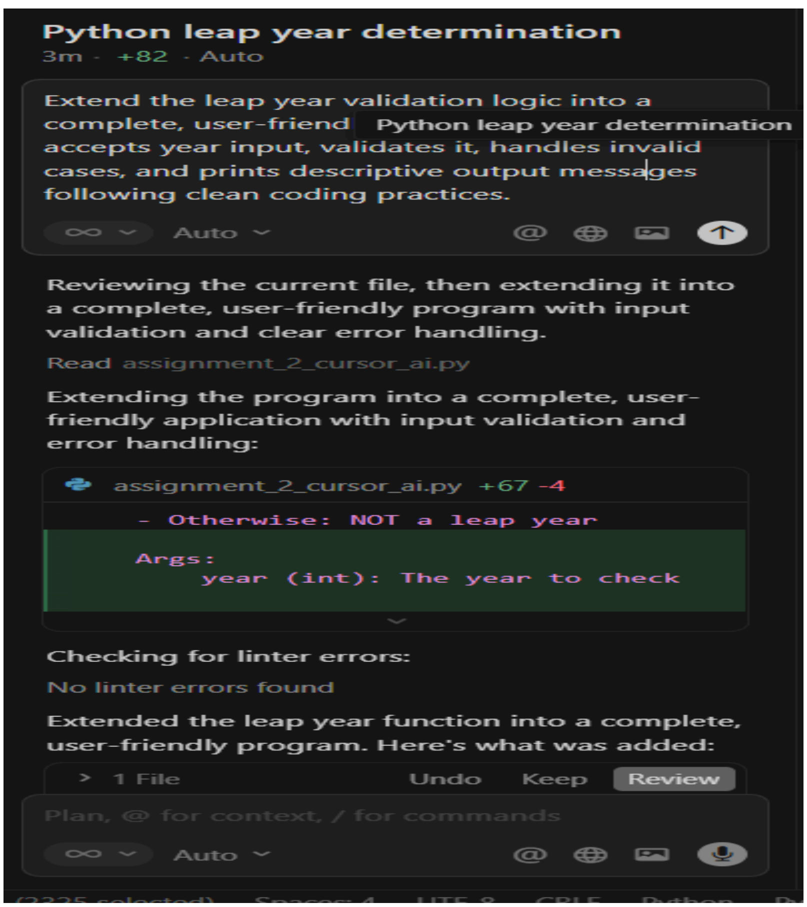
```
Problems    Output    Debug Console    Terminal    Ports                powershell

PS C:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_2.1> & C:\Users\rohit\AppData\Local\Programs\Python\Python314\python.exe c:/Users/rohit/OneDrive/Documents/SRU/ai_code/lab_2.1/assignment_2_cursor_ai.py
2024 is a leap year: True
PS C:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_2.1>

Ctrl+K to generate command
```

Explanation:

The function `is_leap_year` determines whether a given year is a leap year by applying standard calendar rules.

It returns `True` if the year is divisible by 4 but not by 100, or if it is divisible by 400.
A test case is included to validate the function by checking the year 2024 and printing the result.
This approach ensures correctness, simplicity, and reusability of the leap year validation logic.

Refactoring the code:
Prompt:
Extend the leap year validation logic into a complete, user-friendly Python program that accepts year input, validates it, handles invalid cases, and prints descriptive output messages.



Code:

```python
def is_leap_year(year):
    """

    Determine if a given year is a leap year using standard rules.

    Rules:
    - Divisible by 400: leap year
    - Divisible by 100 but not 400: NOT a leap year
    - Divisible by 4 but not 100: leap year
    - Otherwise: NOT a leap year

    Args:
        year (int): The year to check

    Returns:
        bool: True if the year is a leap year, False otherwise
    """
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)


def get_year_input():
    """
    Get and validate year input from the user.

    Returns:
        int: A valid year entered by the user

    Raises:
        ValueError: If the input cannot be converted to an integer
    """
    while True:
        try:
            user_input = input("Enter a year: ").strip()

            if not user_input:
                print("Error: Input cannot be empty. Please try again.")
                continue
```

```
            year = int(user_input)

            if year <= 0:
                print("Error: Year must be a positive integer (greater than 0). Please try again.")
                continue

            return year

        except ValueError:
            print("Error: Invalid input. Please enter a valid integer (e.g., 2024).")
        except KeyboardInterrupt:
            print("\n\nProgram interrupted by user.")
            raise
        except Exception as e:
            print(f"Error: An unexpected error occurred: {e}. Please try again.")


def main():
    """
    Main function to run the leap year validation program.
    """
    print("=" * 50)
    print("Leap Year Validator")
    print("=" * 50)
    print()

    try:
        year = get_year_input()

        if is_leap_year(year):
            print(f"\n✓ {year} is a leap year.")
            print(f"  (It has 366 days with February 29th)")
        else:
            print(f"\n✗ {year} is not a leap year.")
            print(f"  (It has 365 days)")

    except KeyboardInterrupt:
```

```python
        print("\n\nProgram terminated.")
    except Exception as e:
        print(f"\nError: An unexpected error occurred: {e}")


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

Output:

```
PS C:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_2.1> & C:\Users\rohit\AppData\Local\Programs\Python\Python314\python.exe c:/Users/rohit/OneDrive/Documents/SRU/ai_code/lab_2.1/assignment_2_cursor_ai.py
==================================================
Leap Year Validator
==================================================

Enter a year: 2023

✗ 2023 is not a leap year.
  (It has 365 days)
PS C:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_2.1> & C:\Users\rohit\AppData\Local\Programs\Python\Python314\python.exe c:/Users/rohit/OneDrive/Documents/SRU/ai_code/lab_2.1/assignment_2_cursor_ai.py
==================================================
Leap Year Validator
==================================================

Enter a year: 2024
```

```
✓ 2024 is a leap year.
  (It has 366 days with February 29th)
PS C:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_2.1>
```

Explanation:
This program extends the leap year logic into a complete user-interactive application by separating validation, computation, and execution into modular functions.
It improves robustness and usability by handling invalid inputs, providing descriptive output messages, and ensuring clean program flow using structured error handling.

Comparison:

The first implementation focuses on a minimal, function-based approach suitable for backend validation and logical correctness testing.
In contrast, the second implementation expands the logic into a full user-interactive program with input validation, error handling, and descriptive outputs, making it more robust and suitable for real-world applications.

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

Student code:

```
# Student code
numbers = (1, 2, 3, 4, 5, 6)
even_sum = 0
odd_sum = 0

for num in numbers:
```

```python
    if num % 2 == 0:
        even_sum += num
    else:
        odd_sum += num

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

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_2.1> & C:/Users/rohit/AppData/Local
_2.1/assignment_2_git_copilot.py
Sum of even numbers: 12
Sum of odd numbers: 9
PS C:\Users\rohit\OneDrive\Documents\SRU\ai_code\lab_2.1>
```

Explanation:

This code calculates the sum of even and odd numbers in a tuple by iterating through each element and checking its parity using the modulo operator.

Separate variables `even_sum` and `odd_sum` are updated accordingly, and the results are printed, demonstrating clear but verbose manual logic.

Refactoring using AI (Gemini colab):

Prompt:

Refactor the following Python code to improve readability, conciseness, and Pythonic style while preserving its functionality. Please provide the refactored code in a new code cell below the original code

Code:

```python
# Refactored Code (Pythonic and Concise)

numbers = (1, 2, 3, 4, 5, 6)

# Calculate sum of even numbers using a generator expression
even_sum_refactored = sum(num for num in numbers if num % 2 == 0)

# Calculate sum of odd numbers using a generator expression
odd_sum_refactored = sum(num for num in numbers if num % 2 != 0)

print("Refactored - Sum of even numbers:", even_sum_refactored)
print("Refactored - Sum of odd numbers:", odd_sum_refactored)
```

Output:

```
Refactored - Sum of even numbers: 12
Refactored - Sum of odd numbers: 9
```

Explanation:

The AI-refactored code replaces explicit loops and conditional statements with Pythonic generator expressions and the built-in `sum()` function.

This makes the code more concise and readable while preserving the original logic, improving maintainability and following Python best practices.

Comparison:

The original code uses manual iteration and conditionals, which is easy to understand but verbose.

The refactored version achieves the same result in fewer lines, with improved clarity and Pythonic style, making it more elegant and efficient for real-world use.