

# AI Assisted Coding

## Assignment - 02

**Name : A.Rithika**

**Roll No : 2303A54048**

**Batch : 48**

### 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

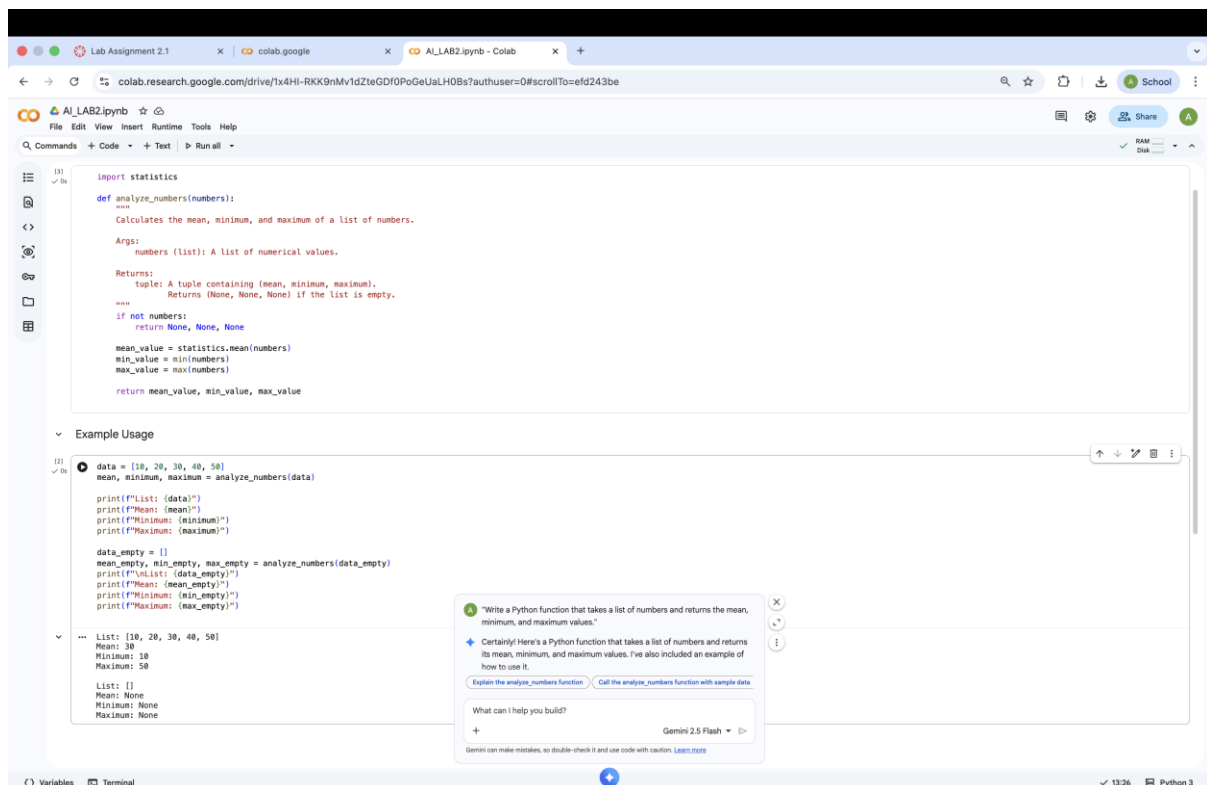
List: [] Mean: None Minimum: None Maximum: None

- **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.



## 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.  
    rgs:  
    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  
 $(1^2 + 0^2 = 1 \neq 10)$

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 displays a Google Colab notebook titled 'AI\_LAB2.ipynb'. The main code cell contains a Python function `is_armstrong_number(number)` with a detailed docstring explaining its purpose, arguments, and return value. The function logic checks if a number is an Armstrong number by summing the cubes of its digits and comparing it to the original number. Below the function, an 'Example Usage' section provides test cases for numbers 153, 9474, 9, 10, and 123, along with their expected boolean results and the underlying calculations. To the right, a Gemini chat window shows the prompt 'Write a Python program to check whether a number is an Armstrong number.' and the generated code snippet.

```
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
```

**Example Usage**

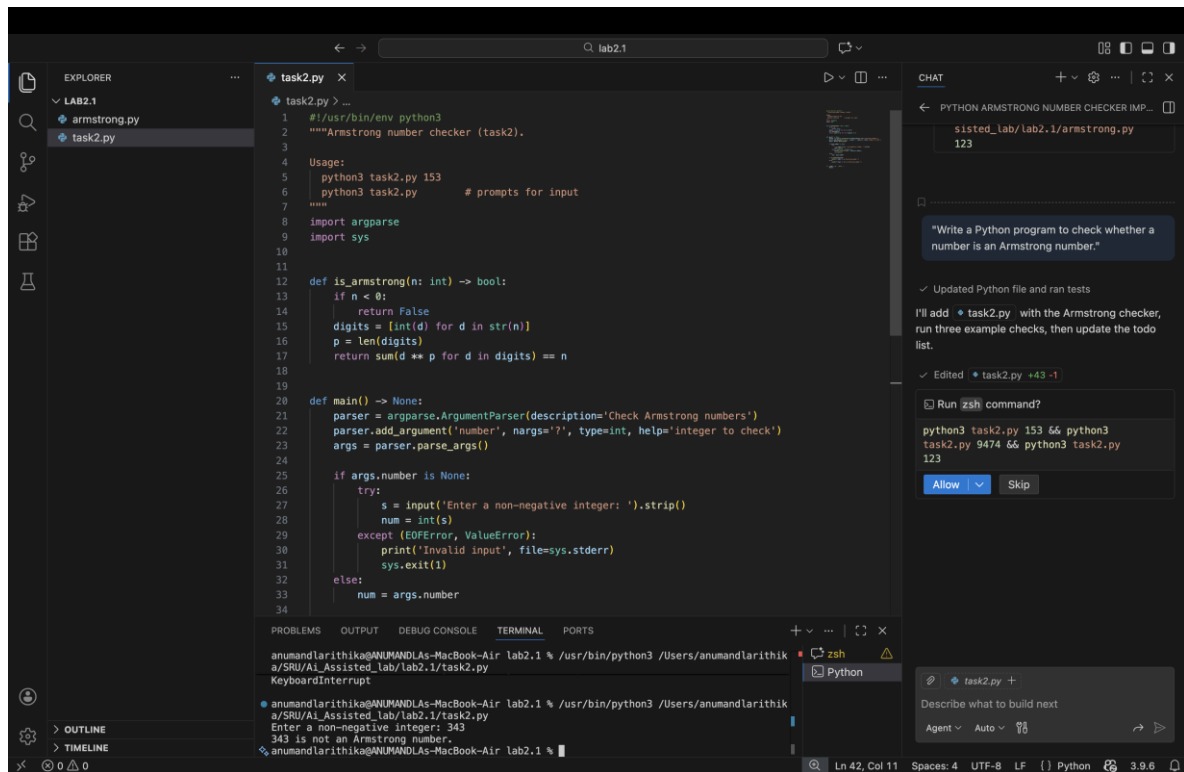
```
# Test with some numbers
print(f"Is 153 an Armstrong number? {is_armstrong_number(153)}") # Expected: True (1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153)
print(f"Is 9474 an Armstrong number? {is_armstrong_number(9474)}") # Expected: True (9^4 + 4^4 + 7^4 + 4^4 = 6561 + 256 + 2401 + 256 = 9474)
print(f"Is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

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

**Gemini Chat:**

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

Okay, here's a Python program that defines a function to check if a number is an Armstrong number, along with some example usage.



## Task 3: Leap Year Validation Using Cursor AI

### ❖ 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 != 0) or (year % 400 == 0):
    print(f'{year} is a leap year.')
else:
    print(f'{year} is not a leap year.')
```

- **Prompt 2 :**

"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 :**

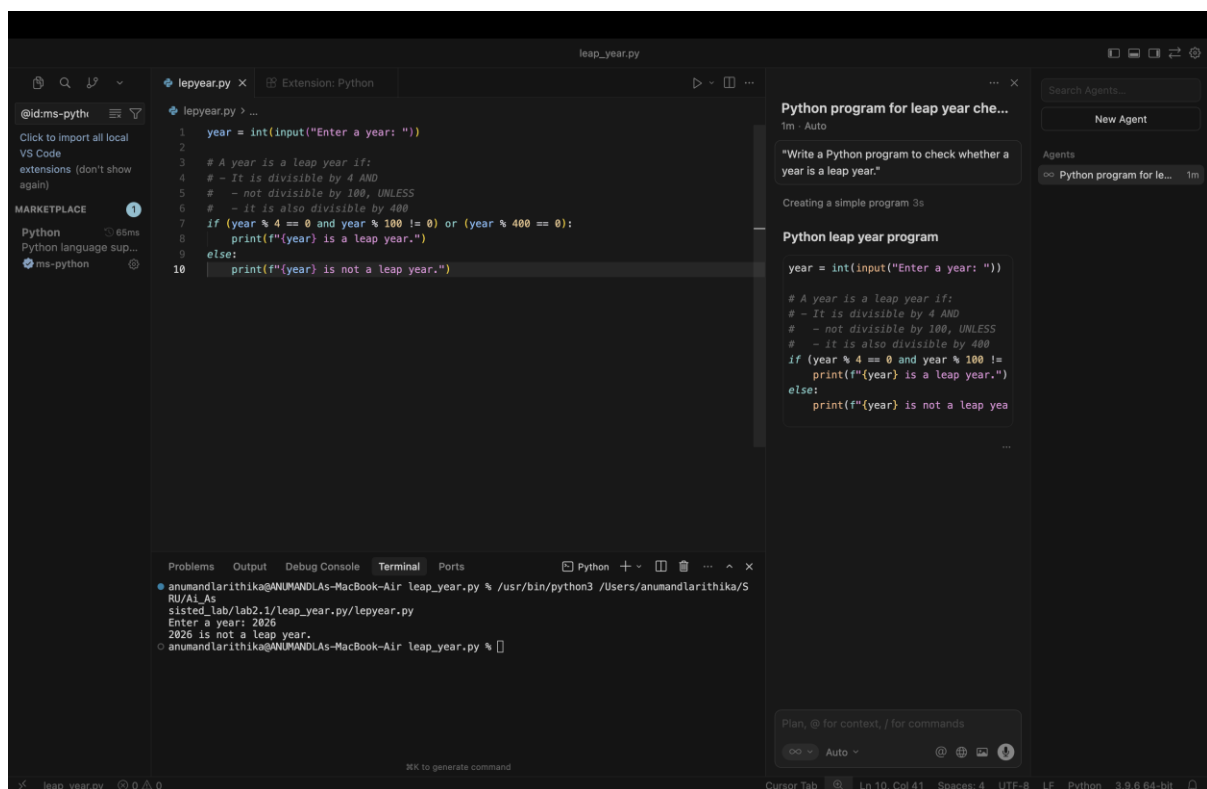
Enter a year: 2026

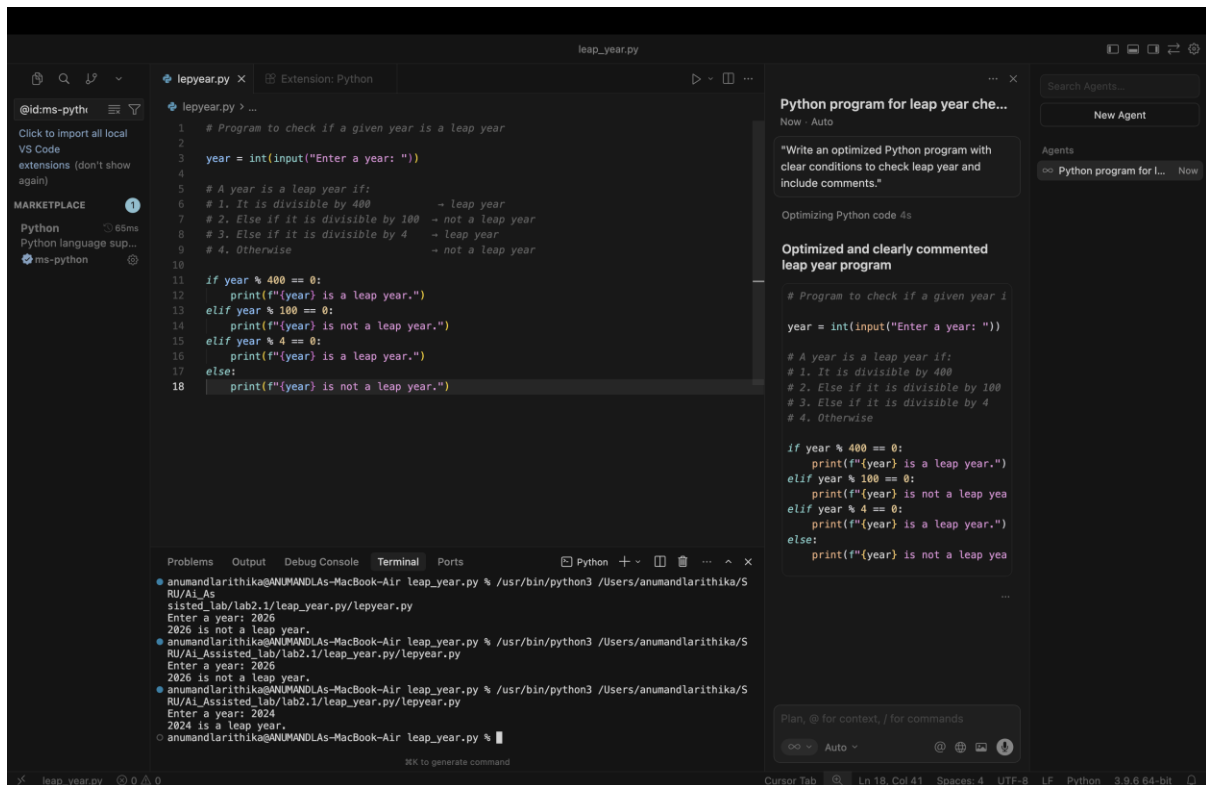
- **Sample Output :**

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.





## 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 displays the Visual Studio Code interface. The main editor window shows a file named `leap_year.py` with the following Python code:

```
1 t = (1, 2, 3, 4, 5, 6, 7)
2
3 # Using generator expressions with sum for clarity and efficiency
4 even_sum = sum(i for i in t if i % 2 == 0)
5 odd_sum = sum(i for i in t if i % 2 != 0)
6
7 print("Sum of even numbers:", even_sum)
8 print("Sum of odd numbers:", odd_sum)
```

The left sidebar shows the 'MARKETPLACE' with 'Python' and 'ms-python' extensions listed. The bottom panel contains the 'Terminal' output, which shows the execution of the script:

```
2026 is not a leap year.
anumandiarithika@ANUMANDLAs-MacBook-Air leap_year.py % /usr/bin/python3 /Users/anumandiarithika/SRU/AI-Assisted_lab/lab2.1/leap_year.py/leap_year.py
Enter a year: 2026
2026 is not a leap year.
anumandiarithika@ANUMANDLAs-MacBook-Air leap_year.py % /usr/bin/python3 /Users/anumandiarithika/SRU/AI-Assisted_lab/lab2.1/leap_year.py/leap_year.py
Enter a year: 2024
2024 is a leap year.
anumandiarithika@ANUMANDLAs-MacBook-Air leap_year.py % /usr/bin/python3 /Users/anumandiarithika/SRU/AI-Assisted_lab/lab2.1/leap_year.py/leap_year.py
Sum of even numbers: 12
Sum of odd numbers: 16
anumandiarithika@ANUMANDLAs-MacBook-Air leap_year.py %
```

On the right side, there is a 'Python program for leap year che...' panel showing a refactored version of the code using generator expressions:

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

even_sum = 0
odd_sum = 0

# Using generator expressions with s
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)
```

The status bar at the bottom indicates the file is `leap_year.py`, line 8, column 38, with 4 spaces, UTF-8 encoding, LF line endings, and Python 3.9.6 64-bit.