AI ASSISTANT CODING

NAME: K. RUCHITHA HT NO:2303A52069 BATCH:32

QUESTION:

Task 1:

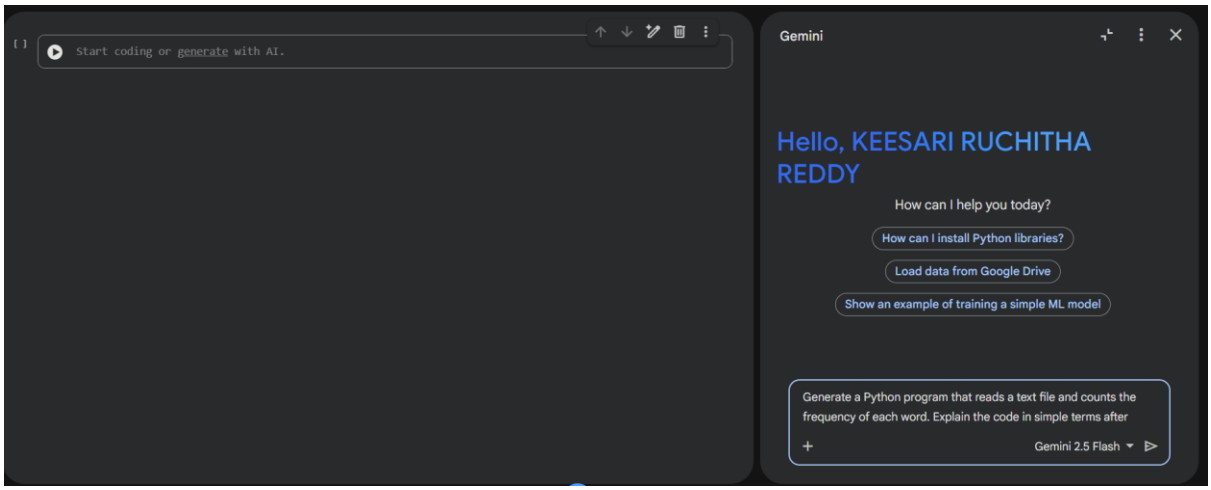
Word Frequency from Text File

Scenario: You are analyzing log files for keyword frequency.

Task: Use Gemini to generate Python code that reads a text file and

counts word frequency, then explains the code

Prompt:



create a Python program that creates a text file, writes sample text into it,

then reads the file and displays its contents.

Add clear comments in the code.

Code Screenshot:

Commands + Code + Text ▷ Run all

[1]

```python
import collections
import re

def count_word_frequency(filepath):
    """
    Reads a text file and returns a dictionary with word frequencies.
    """
    with open(filepath, 'r', encoding='utf-8') as file:
        text = file.read().lower()  # Read content and convert to lowercase
        # Use regular expression to find all words (sequences of letters)
        words = re.findall(r'\b[a-z]+\b', text)
        # Count word frequencies using collections.Counter
        word_counts = collections.Counter(words)
    return word_counts

# Example usage:
# First, let's create a dummy text file for demonstration
file_content = "This is a sample text. This text will be used to count word frequencies. Sample, sample
with open('sample.txt', 'w', encoding='utf-8') as f:
    f.write(file_content)

# Now, call the function with the dummy file
file_path = 'sample.txt'
word_frequencies = count_word_frequency(file_path)

# Display the results
print("Word Frequencies:")
for word, count in word_frequencies.most_common(10):
    print(f"'{word}': {count}")
```

Output:

```
Word Frequencies:
'sample': 3
'text': 3
'this': 2
'is': 1
'a': 1
'will': 1
'be': 1
'used': 1
'to': 1
'count': 1
```

Explanation:

This program reads a text file and splits the content into words.

Each word is stored in a dictionary where the key is the word and the value is its count.

If a word appears again, its count is increased.

Finally, the program prints how many times each word appears in the file.

This helps in analyzing text data and finding frequently used words.

Task 2:

File Operations Using Cursor AI

Scenario: You are automating basic file operations.

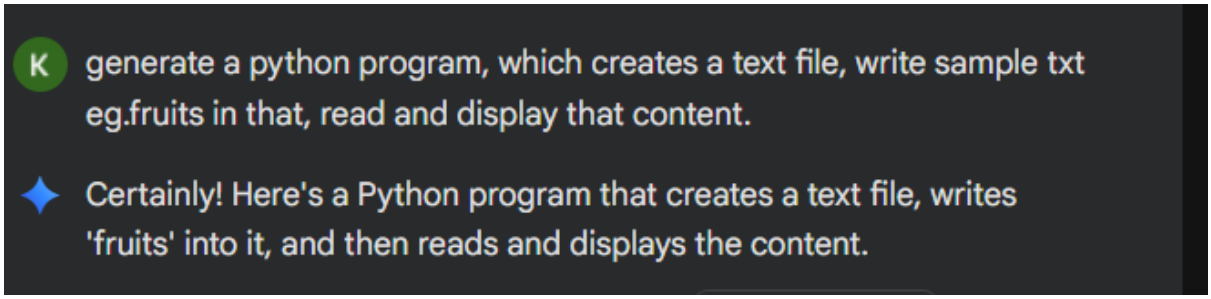
Task: Use Cursor AI to generate a program that:

Creates a text file

Writes sample text

Reads and displays the content

Prompt:



Code screenshot:

```python
# Define the file name
file_name = 'my_fruits.txt'

# Content to write into the file
file_content = "Apple\nBanana\nOrange\nGrape\nStrawberry"

# 1. Create and write to the text file
print(f"Creating file '{file_name}' and writing content...")
with open(file_name, 'w', encoding='utf-8') as file:
    file.write(file_content)
print("Content written successfully.\n")

# 2. Read and display the entire content of the text file
print(f"Reading and displaying entire content from '{file_name}':")
with open(file_name, 'r', encoding='utf-8') as file:
    full_content = file.read()
    print(full_content)
print("\n")

# 3. Re-read and display the content of the text file line by line with line numbers
print(f"Reading and displaying content from '{file_name}' line by line:")
line_number = 1
with open(file_name, 'r', encoding='utf-8') as file:
    for line in file:
        print(f"line {line_number}: {line.strip()}")
        line_number += 1
```

Output:

```
Creating file 'my_fruits.txt' and writing content...
Content written successfully.

Reading and displaying entire content from 'my_fruits.txt':
Apple
Banana
Orange
Grape
Strawberry


Reading and displaying content from 'my_fruits.txt' line by line:
line 1: Apple
line 2: Banana
line 3: Orange
line 4: Grape
line 5: Strawberry
```

Explanation:

This program demonstrates basic file handling in Python using Cursor. First, a text file is created and sample text is written into it. Then, the same file is opened in read mode and its contents are displayed on the screen. It shows how Python can be used to create, write, and read files easily. Such operations are useful in automation and data storage tasks.

Task 3: CSV Data Analysis

Scenario: You are processing structured data from a CSV file.

Task: Use Gemini in Colab to read a CSV file and calculate mean, min,

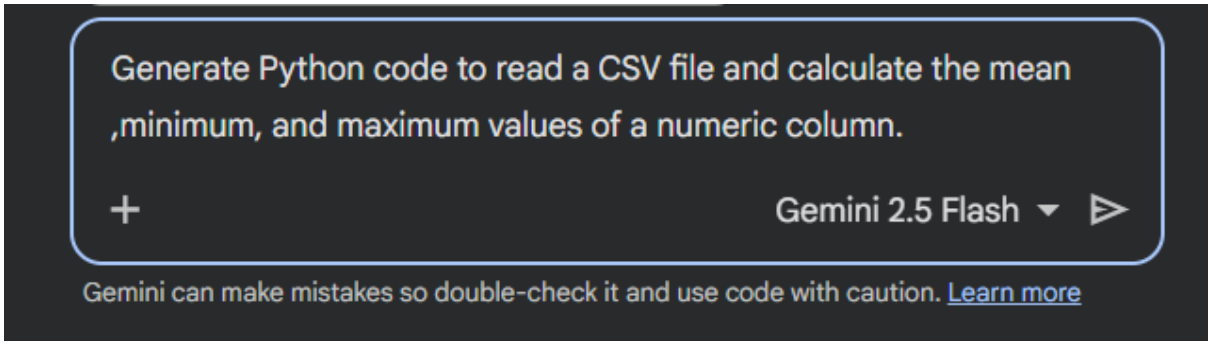
and max.

Prompt:

Generate Python code to read a CSV file and calculate the mean,

minimum, and maximum values of a numeric column.

Prompt:



Code screenshot:

```python
# Define the path to the CSV file
file_path = 'sample_data.csv'

# Define the numeric column to analyze
column_name = 'Score'

# 2. Read the CSV file into a pandas DataFrame
try:
    df = pd.read_csv(file_path)
    print(f"Successfully read '{file_path}'.")
    print("\nDataFrame head:")
    print(df.head())
    print("\n")

    # Check if the column exists in the DataFrame
    if column_name in df.columns:
        # Ensure the column is numeric (errors='coerce' will turn non-numeric into NaN)
        df[column_name] = pd.to_numeric(df[column_name], errors='coerce')

        # Drop rows where the numeric column became NaN due to conversion errors
        df.dropna(subset=[column_name], inplace=True)

        if not df[column_name].empty:
            # 3. Calculate mean, minimum, and maximum values
            mean_value = df[column_name].mean()
            min_value = df[column_name].min()
            max_value = df[column_name].max()

            # 4. Display the results
            print(f"Statistics for column '{column_name}':")
            print(f"  Mean: {mean_value:.2f}")
            print(f"  Minimum: {min_value:.2f}")
            print(f"  Maximum: {max_value:.2f}")
        else:
            print(f"Column '{column_name}' is empty or contains no valid numeric data after cleaning.")
    else:
        print(f"Error: Column '{column_name}' not found in the CSV file.")

except FileNotFoundError:
    print(f"Error: The file '{file_path}' was not found.")
except Exception as e:
    print(f"An error occurred: {e}")
```

Output:

```
Created 'sample_data.csv' with sample data.

Successfully read 'sample_data.csv'.

DataFrame head:
      Name  Age  Score
0    Alice   25     88
1      Bob   30     92
2  Charlie   22     75
3    David   28     95
4      Eve   35     82


Statistics for column 'Score':
  Mean: 86.40
  Minimum: 75.00
  Maximum: 95.00
```

Explanation:

This program reads data from a CSV file using Python.

It extracts numerical values from a column and calculates the mean,

minimum, and maximum.

CSV analysis is used in data processing and analytics applications

Task 4: Sorting Lists Manual vs Built-in

Scenario: You are reviewing algorithm choices for efficiency.

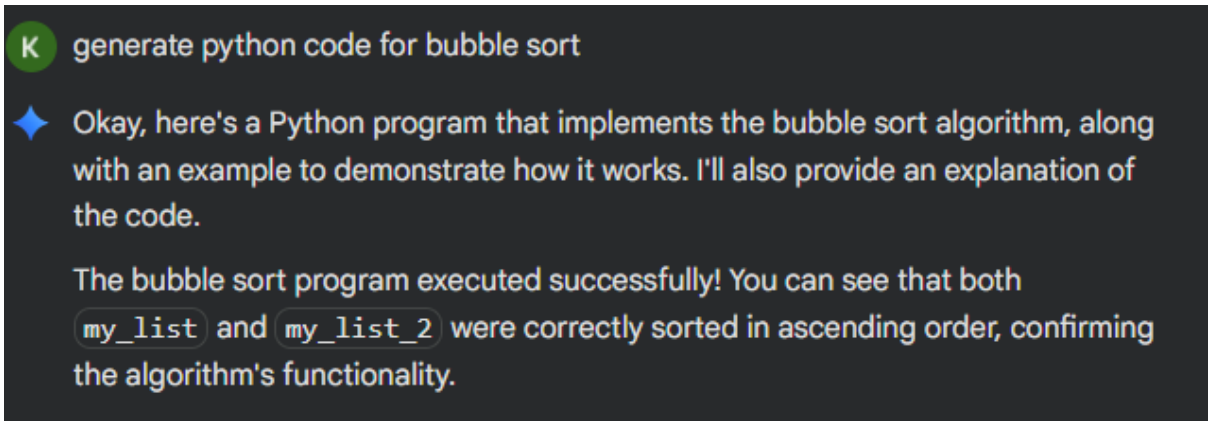
Task: Use Gemini to generate:

Bubble sort

Python's built-in sort()

Compare both implementations.

Prompt:



K generate python code for built in sort

Certainly! Python provides powerful built-in functions for sorting. Here's a program demonstrating `list.sort()` and the `sorted()` function, along with an explanation of each.

The code executed successfully, demonstrating Python's built-in sorting mechanisms!

- `list.sort()` modified `my_list_1` directly, sorting it in-place.
- The `sorted()` function created a *new* sorted list (`sorted_list_2`) from `my_list_2`, leaving `my_list_2` unchanged.
- It also showed how to sort in descending order using `reverse=True` and how to sort a list of strings alphabetically.

Code screenshot:

Manual Bubble Sort Code:

```
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j] # Swap elements
    return arr

# Example usage:
my_list = [64, 34, 25, 12, 22, 11, 90]
print("Original list:", my_list)

sorted_list = bubble_sort(my_list.copy()) # Use .copy() to avoid modifying the original list if you want to reuse it
print("Sorted list:", sorted_list)

my_list_2 = [5, 1, 4, 2, 8]
print("\nOriginal list 2:", my_list_2)
sorted_list_2 = bubble_sort(my_list_2.copy())
print("Sorted list 2:", sorted_list_2)
```

Sorting using sort () function:

```python
# 1. Using the list.sort() method (in-place sort)
print("--- Using list.sort() method ---")
my_list_1 = [64, 34, 25, 12, 22, 11, 90]
print("Original list 1:", my_list_1)
my_list_1.sort() # Sorts the list in-place (modifies the original list)
print("Sorted list 1 (in-place):", my_list_1)

# 2. Using the sorted() built-in function (returns a new sorted list)
print("\n--- Using sorted() function ---")
my_list_2 = [5, 1, 4, 2, 8]
print("Original list 2:", my_list_2)
# sorted() returns a new sorted list, leaving the original unchanged
sorted_list_2 = sorted(my_list_2)
print("Sorted list 2 (new list):", sorted_list_2)
print("Original list 2 (unchanged):", my_list_2)

# Example with reverse order
print("\n--- Sorted in descending order ---")
my_list_3 = [10, 50, 20, 40, 30]
print("Original list 3:", my_list_3)
# Using reverse=True for descending order
sorted_list_3_desc = sorted(my_list_3, reverse=True)
print("Sorted list 3 (descending):", sorted_list_3_desc)

# Sorting a list of strings
print("\n--- Sorting strings ---")
my_strings = ["banana", "apple", "date", "cherry"]
print("Original strings:", my_strings)
sorted_strings = sorted(my_strings)
print("Sorted strings:", sorted_strings)
```

Output:

Bubble sort:

```
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list: [11, 12, 22, 25, 34, 64, 90]

Original list 2: [5, 1, 4, 2, 8]
Sorted list 2: [1, 2, 4, 5, 8]
```

Sort Function:

```
--- Using list.sort() method ---
Original list 1: [64, 34, 25, 12, 22, 11, 90]
Sorted list 1 (in-place): [11, 12, 22, 25, 34, 64, 90]

--- Using sorted() function ---
Original list 2: [5, 1, 4, 2, 8]
Sorted list 2 (new list): [1, 2, 4, 5, 8]
Original list 2 (unchanged): [5, 1, 4, 2, 8]

--- Sorted in descending order ---
Original list 3: [10, 50, 20, 40, 30]
Sorted list 3 (descending): [50, 40, 30, 20, 10]

--- Sorting strings ---
Original strings: ['banana', 'apple', 'date', 'cherry']
Sorted strings: ['apple', 'banana', 'cherry', 'date']
```

Explanation:

Bubble sort is a simple sorting algorithm that repeatedly compares and swaps adjacent elements. It is easy to understand but inefficient for large data sets. Python’s built-in sort function is shorter, optimized and much faster. The built-in method should be preferred in real-world applications.