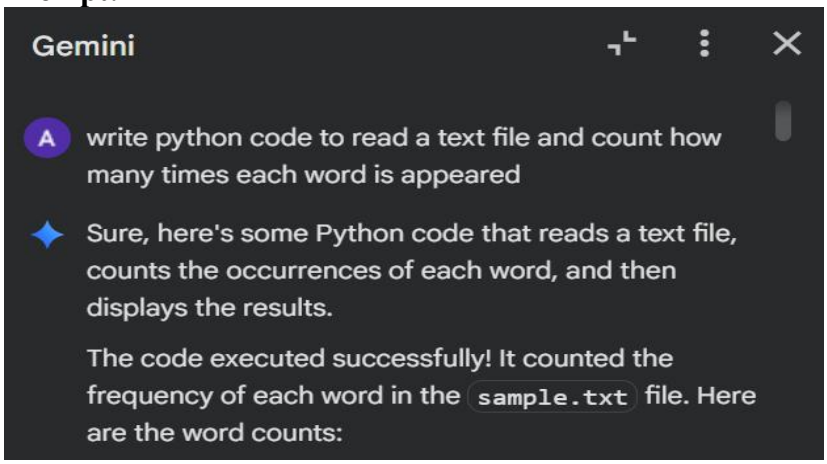
AI ASSISTED CODING

ASSIGNMENT-2.3
2303A52341
B-32

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:




```python
import collections
import re
# Create a dummy text file for demonstration
# This step ensures the code is runnable even if 'sample.txt' doesn't exist
with open('sample.txt', 'w') as f:
    f.write("This is a sample text file. This file contains sample text. Sample is good.")
# Define the path to your text file
# You can change 'sample.txt' to the path of your actual file.
file_path = 'sample.txt'
# Initialize a Counter object to store word frequencies
word_counts = collections.Counter()
# Use a try-except block to handle potential FileNotFoundError or other issues
try:
    # Open the file in read mode ('r') with UTF-8 encoding
    with open(file_path, 'r', encoding='utf-8') as file:
        # Process the file line by line
        for line in file:
            # Convert the line to lowercase to treat words like 'The' and 'the' as the same
            # Use regular expressions to find all words (sequences of word characters)
            words = re.findall(r'\b\w+\b', line.lower())
            # Update the word_counts Counter with the words found in the current line
            word_counts.update(words)
except FileNotFoundError:
    print(f"Error: The file '{file_path}' was not found. Please check the file path.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
# Display the word counts
# .most_common() returns a list of (word, count) tuples, sorted by count (descending)
print("Word frequencies:")
for word, count in word_counts.most_common():
    print(f"{word}: {count}")
```

Output(O/P):

```
Word frequencies:
sample: 3
this: 2
is: 2
text: 2
file: 2
a: 1
contains: 1
good: 1
```

Explanation:

This program first creates a small text file so we have something to work with.
It then opens the file and reads each line, turning everything into lowercase so words like Sample and sample are counted the same.
Using a regular expression, it picks out all the words from the text.
Python tool called **Counter** keeps track of how many times each word shows up.
If the file is missing or something goes wrong, the program shows a clear error message instead of crashing.
Finally, it prints the words along with their counts, starting from the most frequent ones.

Task 2:

File Operations Using Cursor AI

Scenario:

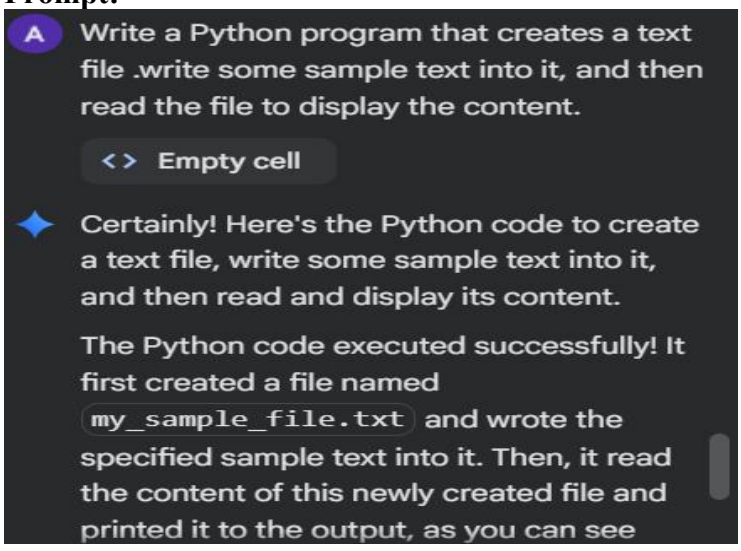
You are automating basic file operations.

Task:

Use Cursor AI to generate a program that:

Creates a text file
Writes sample text
Reads and displays the content

Prompt:



```python
# Define the filename
file_name = 'my_sample_file.txt'

# Sample text to write into the file
sample_text = (
    "This is the first line of sample text.\n"
    "This is the second line.\n"
    "And finally, the third line."
)

# 1. Create a text file and write sample text into it
try:
    with open(file_name, 'w', encoding='utf-8') as file:
        file.write(sample_text)
    print(f"Successfully created '{file_name}' and wrote content.")
except IOError as e:
    print(f"Error writing to file '{file_name}': {e}")

# 2. Read the file to display its content
try:
    with open(file_name, 'r', encoding='utf-8') as file:
        content = file.read()
    print(f"\nContent of '{file_name}':\n---\n{content}\n---")
except FileNotFoundError:
    print(f"Error: The file '{file_name}' was not found.")
except IOError as e:
    print(f"Error reading from file '{file_name}': {e}")
```

Output(O/P):

```
Successfully created 'my_sample_file.txt' and wrote content.

Content of 'my_sample_file.txt':
---
This is the first line of sample text.
This is the second line.
And finally, the third line.
---
```

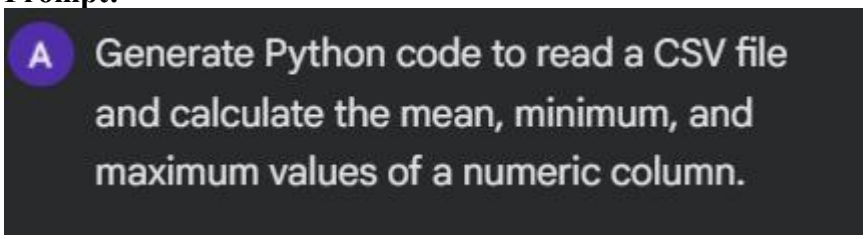
Explanation:

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.
Expected Output:
Correct output Screenshot

Prompt:



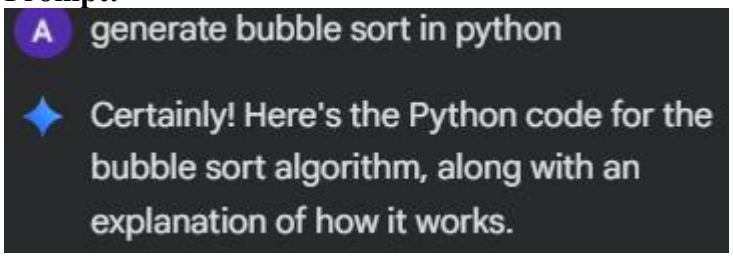

```python
import pandas as pd
import io
# Create a dummy CSV file for demonstration
csv_data = "col1,col2,numeric_column,col4\n" \
           "A,10,100.5,X\n" \
           "B,20,150.0,Y\n" \
           "C,30,99.9,Z\n" \
           "D,40,200.1,W\n" \
           "E,50,125.7,V"
file_name = 'sample.csv'
with open(file_name, 'w') as f:
    f.write(csv_data)
print(f"Created '{file_name}' with sample data.")
try:
    # Read the CSV file into a pandas DataFrame
    df = pd.read_csv(file_name)
    print(f"\nContent of '{file_name}':\n")
    print(df)
    # Specify the numeric column you want to analyze
    numeric_column_name = 'numeric_column'
    # Check if the column exists and is numeric
    if numeric_column_name in df.columns and pd.api.types.is_numeric_dtype(df[numeric_column_name]):
        # Calculate mean, minimum, and maximum
        mean_value = df[numeric_column_name].mean()
        min_value = df[numeric_column_name].min()
        max_value = df[numeric_column_name].max()
        print(f"\nAnalysis for column '{numeric_column_name}':")
        print(f"  Mean: {mean_value:.2f}")
        print(f"  Minimum: {min_value:.2f}")
        print(f"  Maximum: {max_value:.2f}")
    else:
        print(f"Error: Column '{numeric_column_name}' not found or is not numeric.")
except FileNotFoundError:
    print(f"Error: The file '{file_name}' was not found. Please check the file path.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

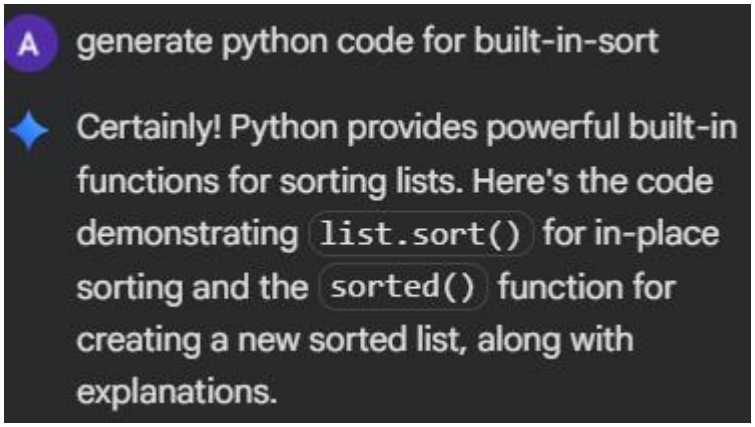
Output(O/P):

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

Content of 'sample.csv':

  col1  col2  numeric_column col4
0    A    10           100.5    X
1    B    20           150.0    Y
2    C    30            99.9    Z
3    D    40           200.1    W
4    E    50           125.7    V

Analysis for column 'numeric_column':
  Mean: 135.24
  Minimum: 99.90
  Maximum: 200.10
```

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.
Expected Output:
Two versions of code
Short comparison
Prompt:





```python
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # Last i elements are already in place
        # So we only need to compare up to n-i-1
        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(f"Original list: {my_list}")

sorted_list = bubble_sort(my_list)
print(f"Sorted list: {sorted_list}")

my_list_2 = [5, 1, 4, 2, 8]
print(f"Original list: {my_list_2}")
sorted_list_2 = bubble_sort(my_list_2)
print(f"Sorted list: {sorted_list_2}")
```

Output(O/P):

```
Original list: [64, 34, 25, 12, 22, 11, 90]
Sorted list: [11, 12, 22, 25, 34, 64, 90]
Original list: [5, 1, 4, 2, 8]
Sorted list: [1, 2, 4, 5, 8]
```

```python
# Example 1: Using the list.sort() method (sorts in-place)
my_list_1 = [64, 34, 25, 12, 22, 11, 90]
print(f"Original list 1: {my_list_1}")
my_list_1.sort()
print(f"Sorted list 1 (in-place using .sort()): {my_list_1}")

print("\n---")

# Example 2: Using the sorted() built-in function (returns a new sorted list)
my_list_2 = [5, 1, 4, 2, 8]
print(f"Original list 2: {my_list_2}")
sorted_list_2 = sorted(my_list_2)
print(f"Sorted list 2 (new list using sorted()): {sorted_list_2}")
print(f"Original list 2 remains unchanged: {my_list_2}")

print("\n---")

# Example 3: Sorting with a custom key (e.g., sort by length of strings)
words = ["apple", "banana", "kiwi", "grapefruit", "fig"]
print(f"Original words list: {words}")
sorted_by_length = sorted(words, key=len)
print(f"Sorted words by length: {sorted_by_length}")

# Example 4: Sorting in descending order
numbers = [10, 30, 20, 50, 40]
print(f"Original numbers list: {numbers}")
sorted_descending = sorted(numbers, reverse=True)
print(f"Sorted numbers in descending order: {sorted_descending}")
```

Output(O/P):

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

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

---
Original words list: ['apple', 'banana', 'kiwi', 'grapefruit', 'fig']
Sorted words by length: ['fig', 'kiwi', 'apple', 'banana', 'grapefruit']
Original numbers list: [10, 30, 20, 50, 40]
Sorted numbers in descending order: [50, 40, 30, 20, 10]
```

Explanation:

Bubble sort is a very basic sorting method. It works by comparing two consecutive elements and swap them if they are in the wrong order. This process repeats many times until the list is sorted.

Python's built- in sort() function is much faster and more efficient. which is optimized for real- world data. The built- in method is shorter to write, runs quickly, and should be preferred in practical applications.