AI Assisted Coding

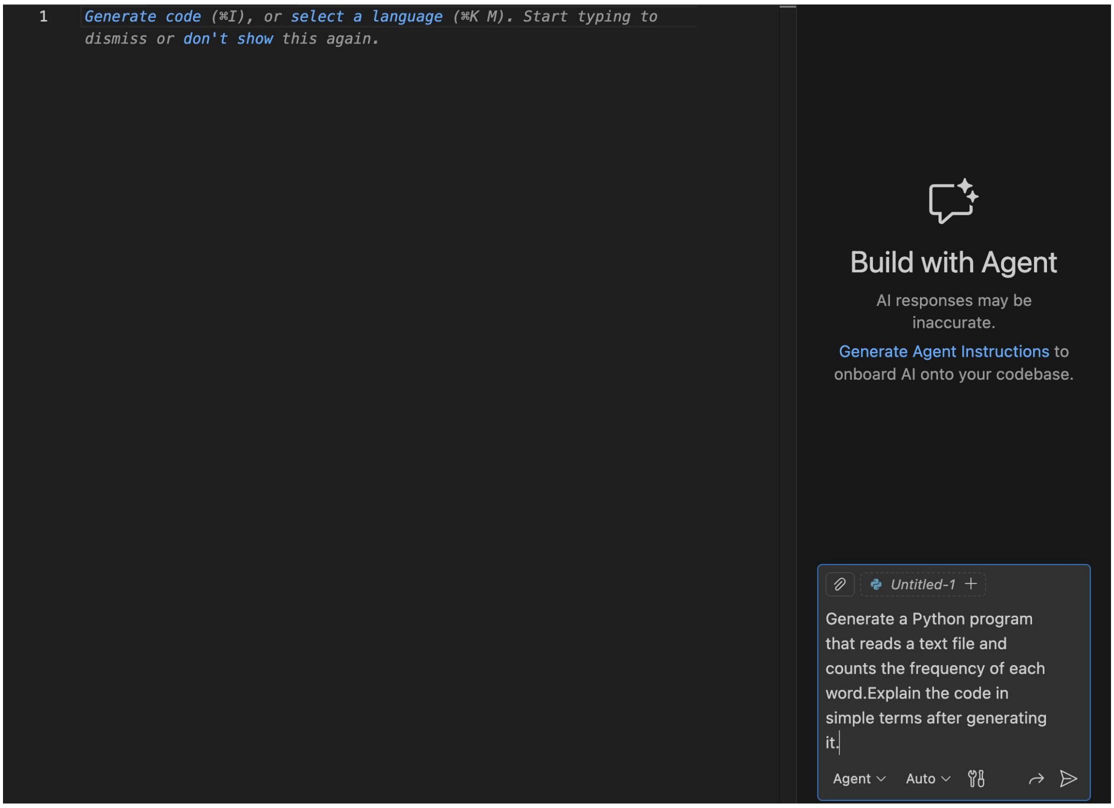
Assignment 2.3

Name:T.Akshaya
HT No: 2303A52017
Batch:31

Question:

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:

Files

- ..
- sample_data
- words.txt

```python
import re
from collections import Counter

def count_word_frequencies(filepath):
    """
    Reads a text file, counts the frequency of each word, and returns the counts.
    Words are converted to lowercase and punctuation is ignored.
    """
    try:
        with open(filepath, 'r') as file:
            text = file.read()
            # Convert text to lowercase and find all word sequences (alphanumeric)
            # \b matches word boundaries, \w+ matches one or more word characters
            words = re.findall(r'\b\w+\b', text.lower())

            # Use collections.Counter to easily count word occurrences
            word_counts = Counter(words)
            return word_counts
    except FileNotFoundError:
        return f"Error: The file '{filepath}' was not found. Please ensure it exists."
    except Exception as e:
        return f"An unexpected error occurred: {e}"

# --- Main execution ---
# Assuming the text file is named 'words.txt' and is located in the '/content/' directory.
# You can change 'file_path' if your file is named differently or in another location.
file_path = '/content/words.txt'

print(f"Reading file: {file_path}\n")

frequencies = count_word_frequencies(file_path)

# Check if the function returned a dictionary (Counter object) or an error string
if isinstance(frequencies, dict):
    print("Word Frequencies (most common first):")
    # Iterate through the most common words and their counts
    for word, count in frequencies.most_common():
        print(f"  '{word}': {count}")
else:
    print(frequencies) # Print the error message
```

output:

```
    print(frequencies) # Print the e

Reading file: /content/words.txt

Word Frequencies (most common first):
  'apple': 4
  'banana': 3
  'orange': 3
  'dog': 3
  'cat': 2
```

Explanation:

The program reads an input text file and performs tokenization to extract individual words. It utilizes a dictionary where each unique word serves as a key and its corresponding value represents the frequency count. As the file is processed, the program checks for the existence of each word in the dictionary; if the word already exists, its count is incremented, otherwise a new entry is created. After processing the entire file, the program outputs the frequency distribution of all words, enabling quantitative analysis of word usage within the text.

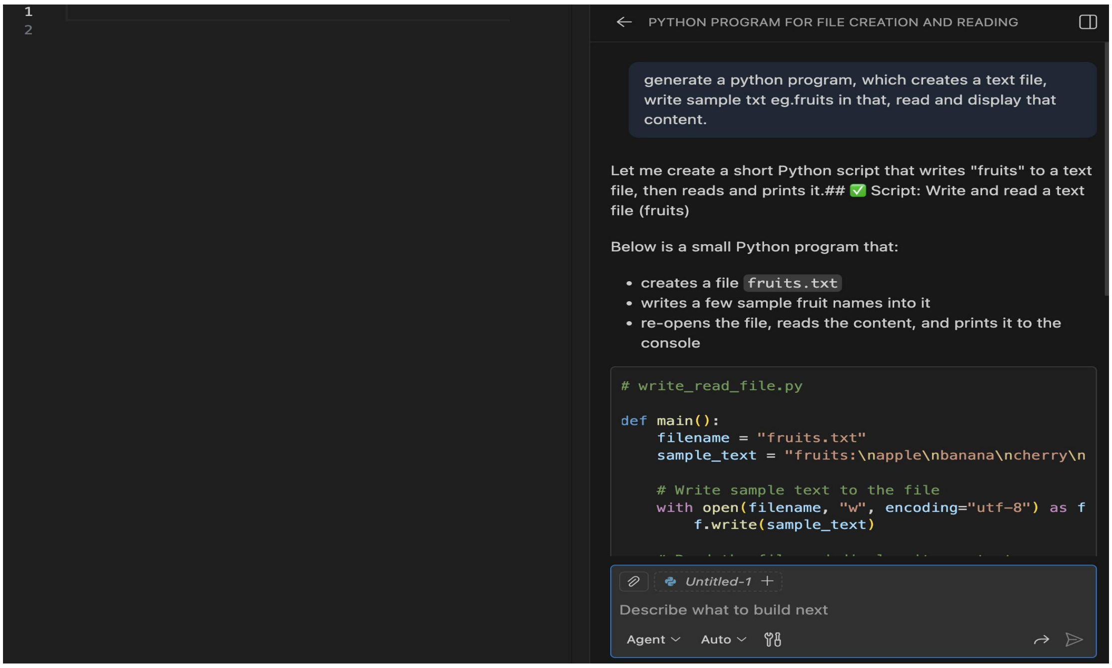
Question:

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

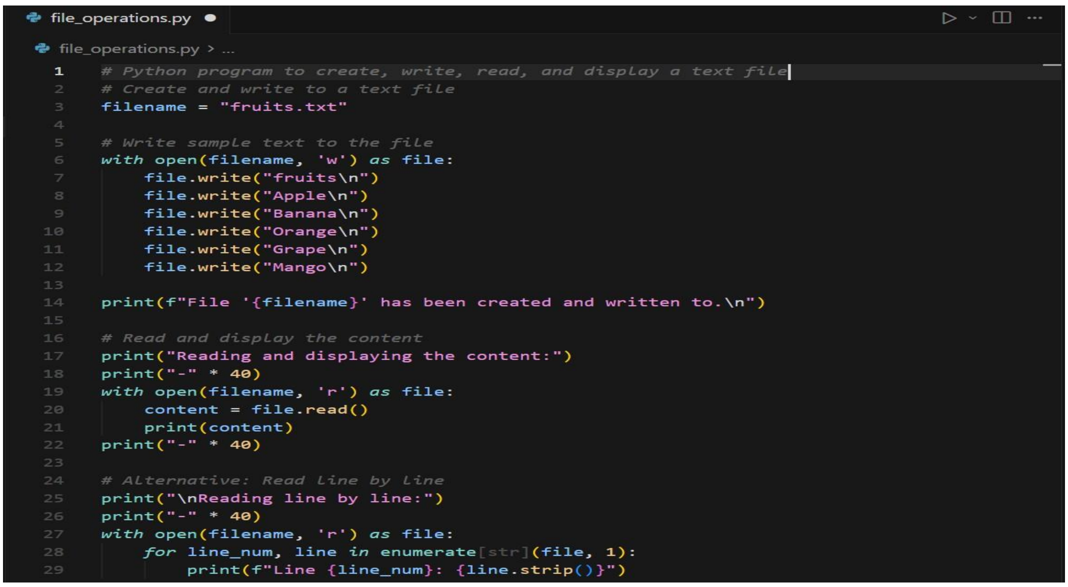
Reads and displays the content

Prompt:

generate a python program, which creates a text file, write sample txt eg.fruits in that, read and display that content.



Code:



```
# Python program to create, write, read, and display a text file
# Create and write to a text file
filename = "fruits.txt"

# Write sample text to the file
with open(filename, 'w') as file:
    file.write("fruits\n")
    file.write("Apple\n")
    file.write("Banana\n")
    file.write("Orange\n")
    file.write("Grape\n")
    file.write("Mango\n")

print(f"File '{filename}' has been created and written to.\n")

# Read and display the content
print("Reading and displaying the content:")
print("-" * 40)
with open(filename, 'r') as file:
    content = file.read()
    print(content)
print("-" * 40)

# Alternative: Read line by line
print("\nReading line by line:")
print("-" * 40)
with open(filename, 'r') as file:
    for line_num, line in enumerate(file, 1):
        print(f"Line {line_num}: {line.strip()}")
```

Output:

```
Problems   Output   Debug Console   Terminal   Ports

$ python file_operations.py
File 'fruits.txt' has been created and written to.

Reading and displaying the content:
----------------------------------------
fruits
Apple
Banana
Orange
Grape
Mango

----------------------------------------

Reading line by line:
----------------------------------------
Line 1: fruits
Line 2: Apple
Line 3: Banana
Line 4: Orange
Line 5: Grape
Line 6: Mango
$ 
```

Explanation:

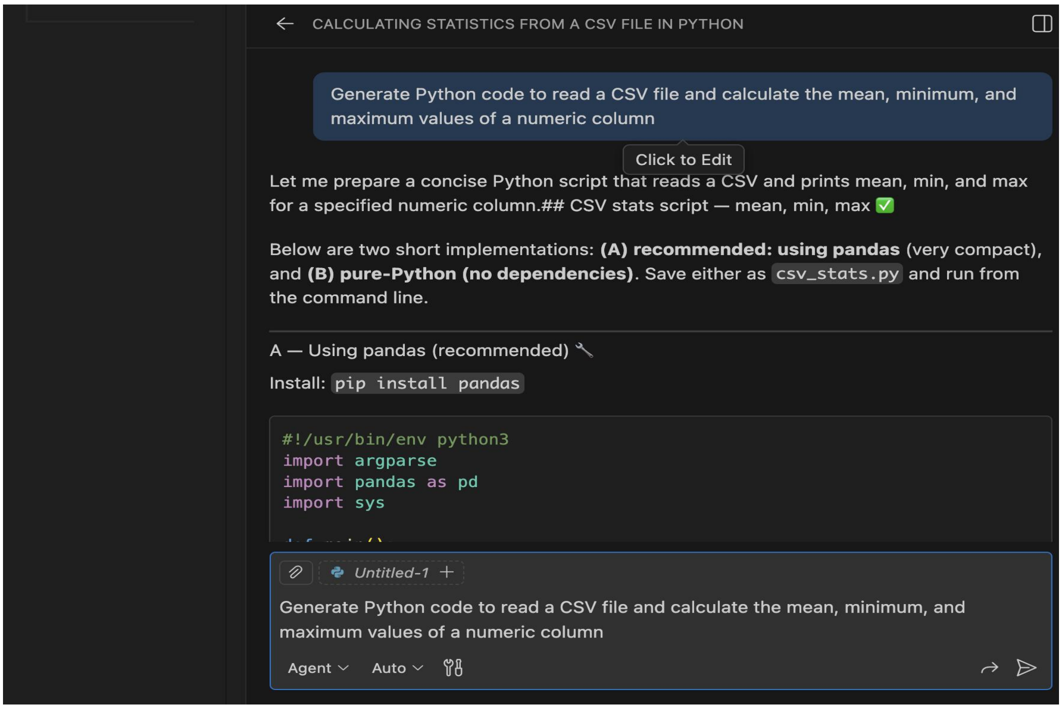
This program illustrates fundamental file input/output operations in Python using a file cursor (file object). Initially, a text file is created and opened in write mode, allowing sample data to be written to persistent storage. The file is then closed and reopened in read mode, where the cursor sequentially accesses the file contents and displays them on the output screen. This demonstrates Python's capability to handle file creation, writing, and reading operations efficiently. Such file-handling mechanisms are essential in automation workflows, data persistence, and log management systems.

Question:

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:

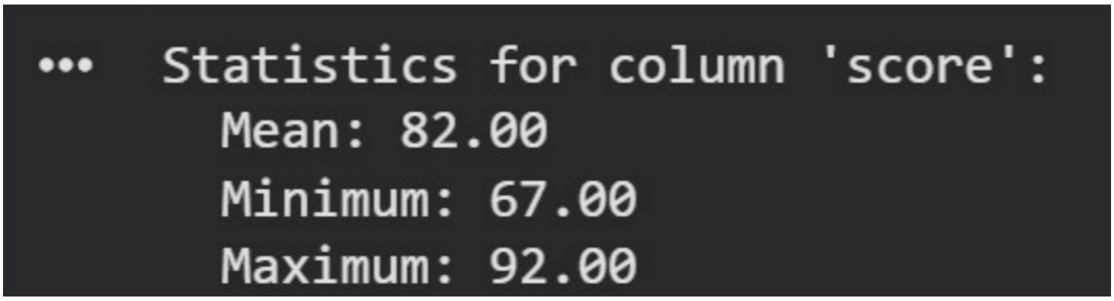
Generate Python code to read a CSV file and calculate the mean, minimum, and maximum values of a numeric column



Output:

```python
import pandas as pd
import io

# Uncomment the lines below and replace 'your_file.csv' with your CSV file path
try:
    df = pd.read_csv('scores.csv')
except FileNotFoundError:
    print("Error: CSV file not found. Please check the path.")
    exit()

# Specify the numeric column you want to analyze
column_name = 'score'

# Check if the column exists in the DataFrame
if column_name in df.columns:
    # Ensure the column is numeric (e.g., float or int)
    # pd.to_numeric will convert values to numeric, coercing errors to NaN
    numeric_column = pd.to_numeric(df[column_name], errors='coerce')

    # Drop rows where the numeric conversion resulted in NaN (non-numeric values)
    numeric_column = numeric_column.dropna()

    if not numeric_column.empty:
        # Calculate mean, minimum, and maximum
        mean_value = numeric_column.mean()
        min_value = numeric_column.min()
        max_value = numeric_column.max()

        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}' contains no valid numeric data after cleaning.")
else:
    print(f"Error: Column '{column_name}' not found in the CSV file. Available columns: {df.columns.tolist()}")
```

Output:



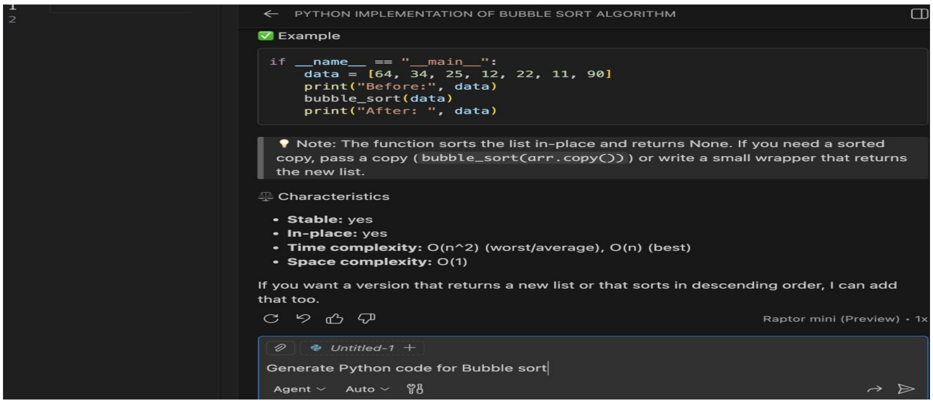
Explanation:

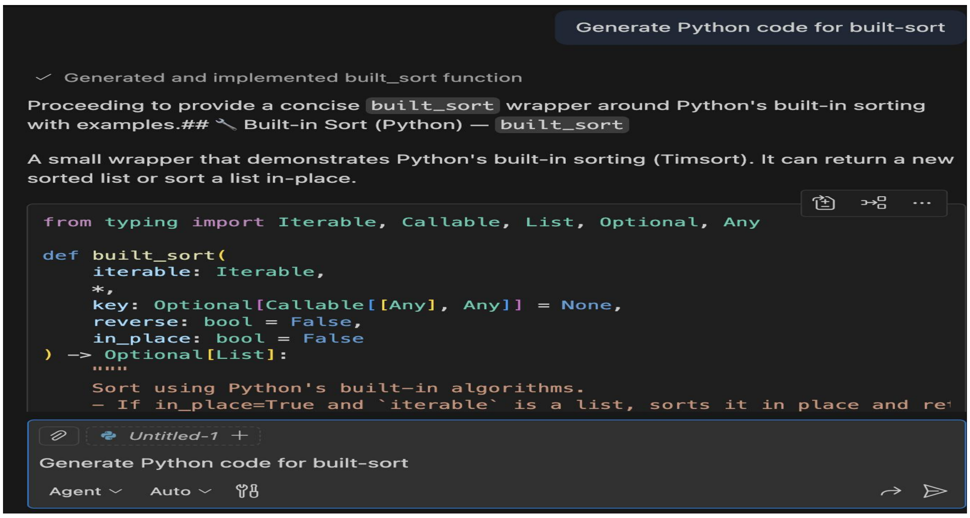
This program performs data extraction and analysis from a CSV file using Python. It reads structured tabular data, retrieves numerical values from a specified column, and computes basic statistical measures such as the mean, minimum, and maximum. This type of CSV data processing is widely used in data analytics, reporting systems, and preprocessing tasks for data-driven applications.

Question:

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:





Code:

Manual Bubble Sort Code:

```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
        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)
print("Sorted list using Bubble Sort:", 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)
print("Sorted list 2 using Bubble Sort:", sorted_list_2)
```

Sorting using sort() function:

```python
# --- Using the list.sort() method ---
# This method sorts the list in-place (modifies the original list) and returns None.

my_list_1 = [64, 34, 25, 12, 22, 11, 90]
print("Original list (list.sort()):", my_list_1)

my_list_1.sort()
print("Sorted list (list.sort()):", my_list_1)

# You can also sort in descending order
my_list_2 = ['banana', 'apple', 'cherry', 'date']
print("\nOriginal list (descending):", my_list_2)

my_list_2.sort(reverse=True)
print("Sorted list (descending):", my_list_2)
```

Output:

Bubble sort:

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

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

Sort Function:

```
Original list (list.sort()): [64, 34, 25, 12, 22, 11, 90]
Sorted list (list.sort()): [11, 12, 22, 25, 34, 64, 90]

Original list (descending): ['banana', 'apple', 'cherry', 'date']
Sorted list (descending): ['date', 'cherry', 'banana', 'apple']
```

Explanation:

Bubble sort is a comparison-based sorting algorithm that repeatedly iterates through a list, comparing adjacent elements and swapping them if they are in the incorrect order. This process continues until the list is fully sorted. Although the algorithm is straightforward and easy to implement, it has a time complexity of O(n2), making it inefficient for large datasets. In contrast, Python's built-in sort() function uses highly optimized algorithms that provide significantly better performance and scalability. Therefore, the built-in sorting method is recommended for real-world and production-level applications.