AI ASSISTED CODING

ASSIGNMENT 2.3

Name: Madadi Uday

Ht No: 2303A52038

Batch – 31

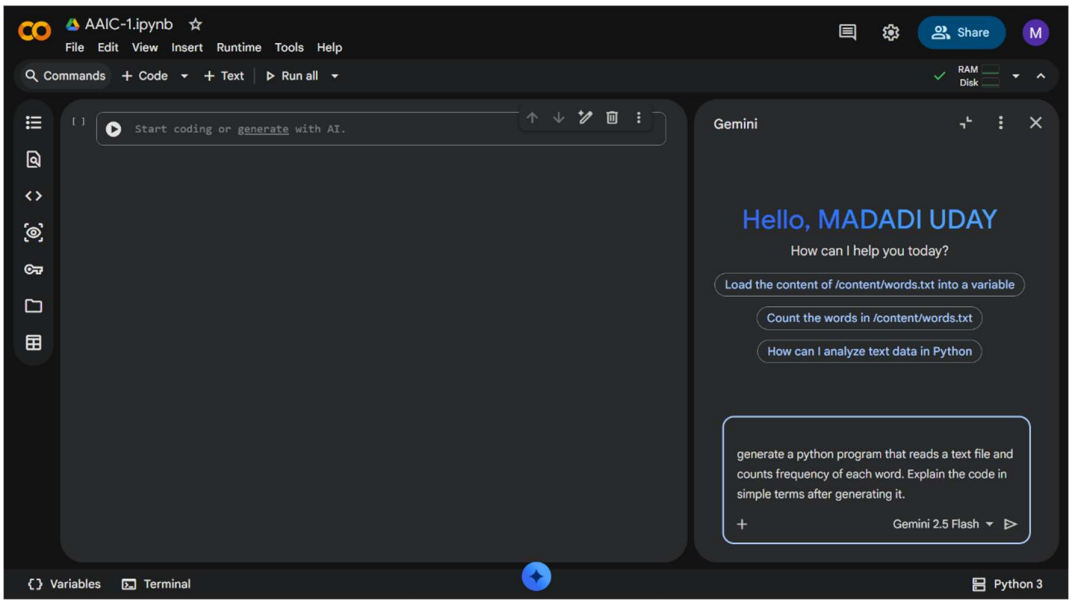
Task 1:

Word Frequency from Text File

Scenario: You are analyzing log files for keyword frequency.

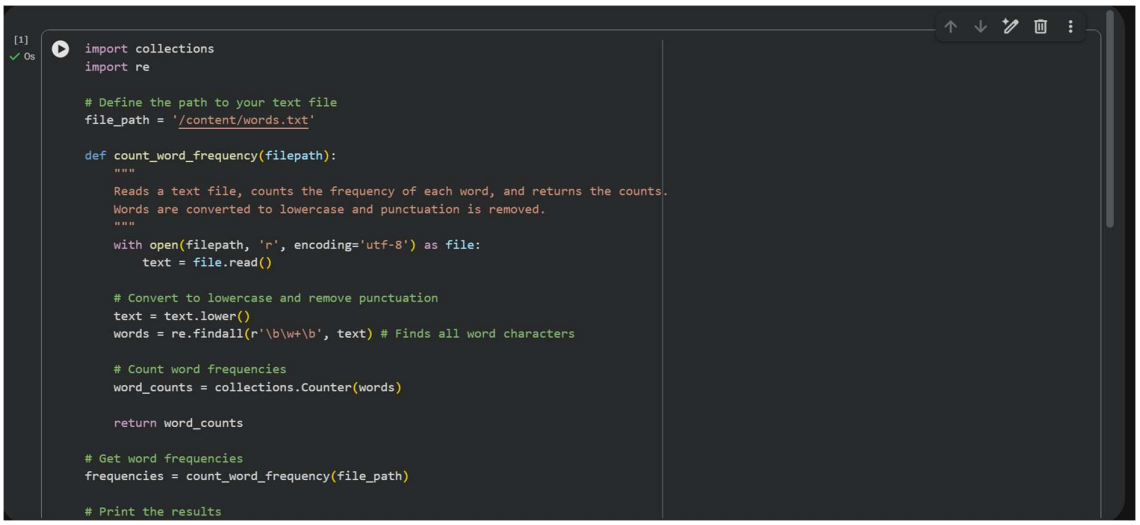
Task: Use Gemini to generate Python code that reads a text file and counts word frequency, then explains the code

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:



```
[1]
✓ 0s
import collections
import re

# Define the path to your text file
file_path = '/content/words.txt'

def count_word_frequency(filepath):
    """
    Reads a text file, counts the frequency of each word, and returns the counts.
    Words are converted to lowercase and punctuation is removed.
    """
    with open(filepath, 'r', encoding='utf-8') as file:
        text = file.read()

    # Convert to lowercase and remove punctuation
    text = text.lower()
    words = re.findall(r'\b\w+\b', text) # Finds all word characters

    # Count word frequencies
    word_counts = collections.Counter(words)

    return word_counts

# Get word frequencies
frequencies = count_word_frequency(file_path)

# Print the results
```

OUTPUT(O/P):

```
# Print the results
print("Word Frequencies:")
for word, count in frequencies.most_common(10): # Display top 10 most common words
    print(f"'{word}': {count}")

•••  Word Frequencies:
'apple': 4
'banana': 3
'orange': 3
'dog': 3
'cat': 2
```

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

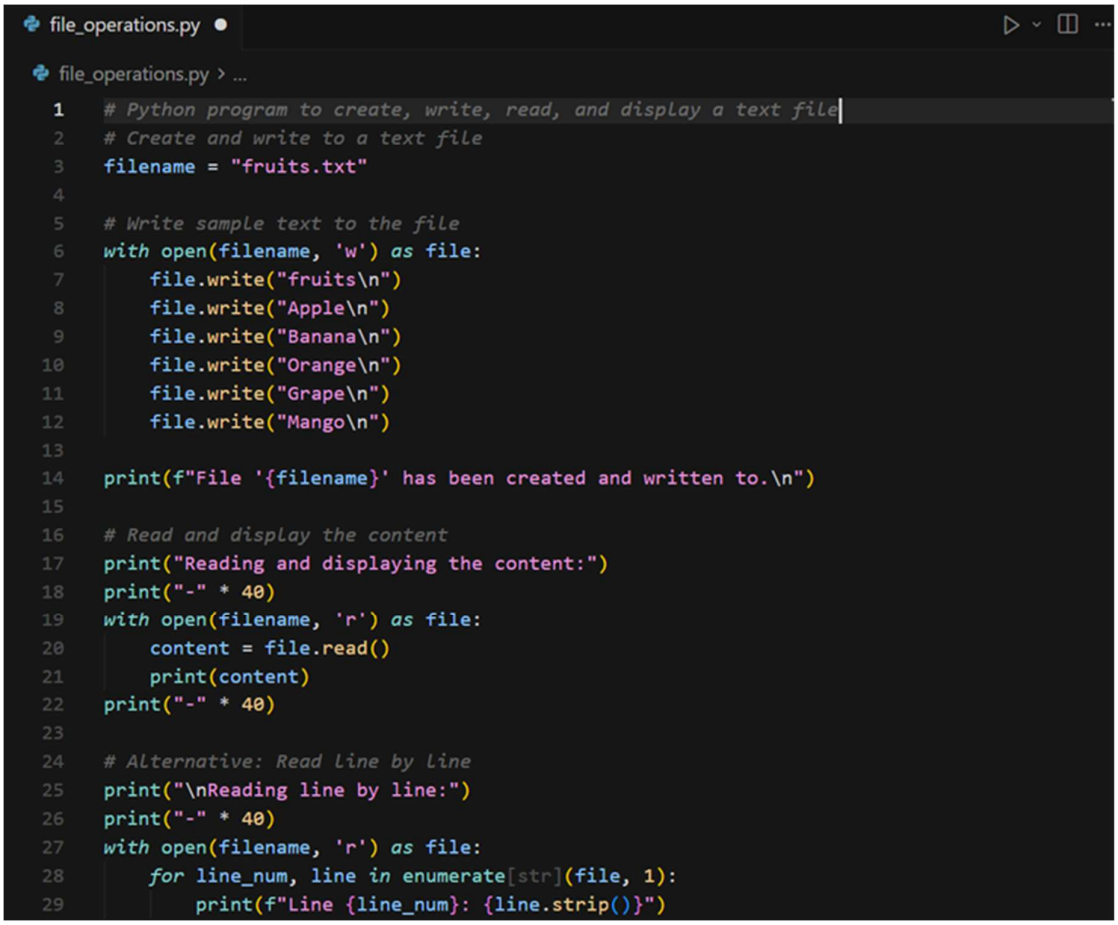
~Writes sample text

~Reads and displays the content

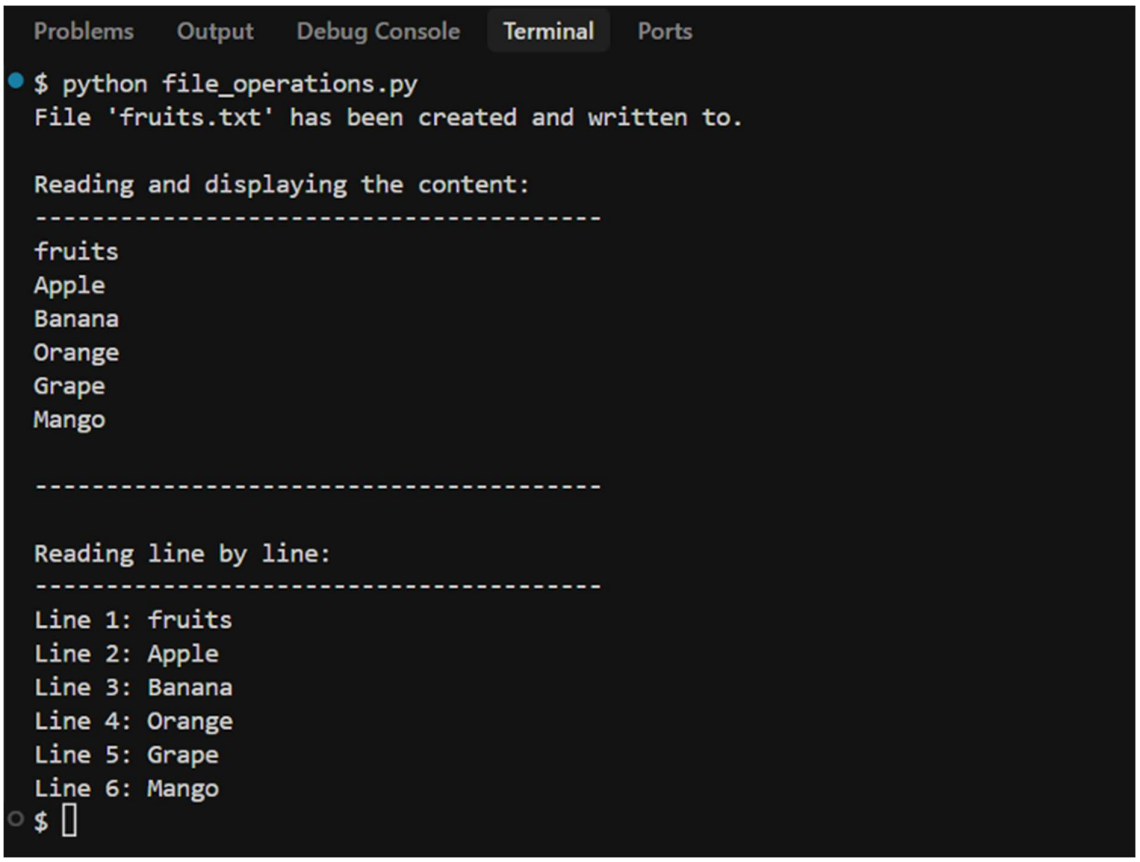
Prompt:

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

Code Screenshot:



Output(O/P):



Explanation:

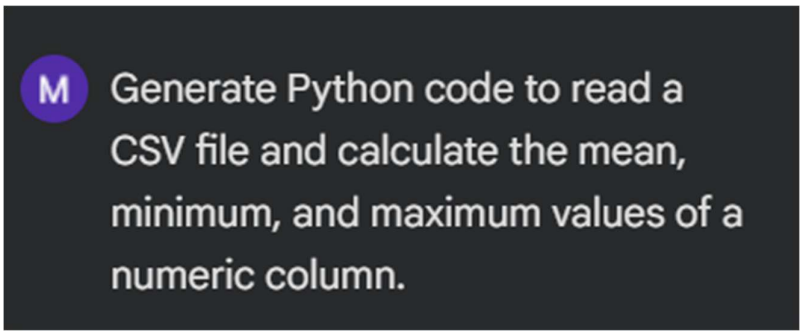
This program demonstrates basic file handling in Python using Cursor. First, a text file is created and sample text is written into it. Then, the same file is opened in read mode and its contents are displayed on the screen. It shows how Python can be used to create, write, and read files easily. Such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

Task 3: CSV Data Analysis

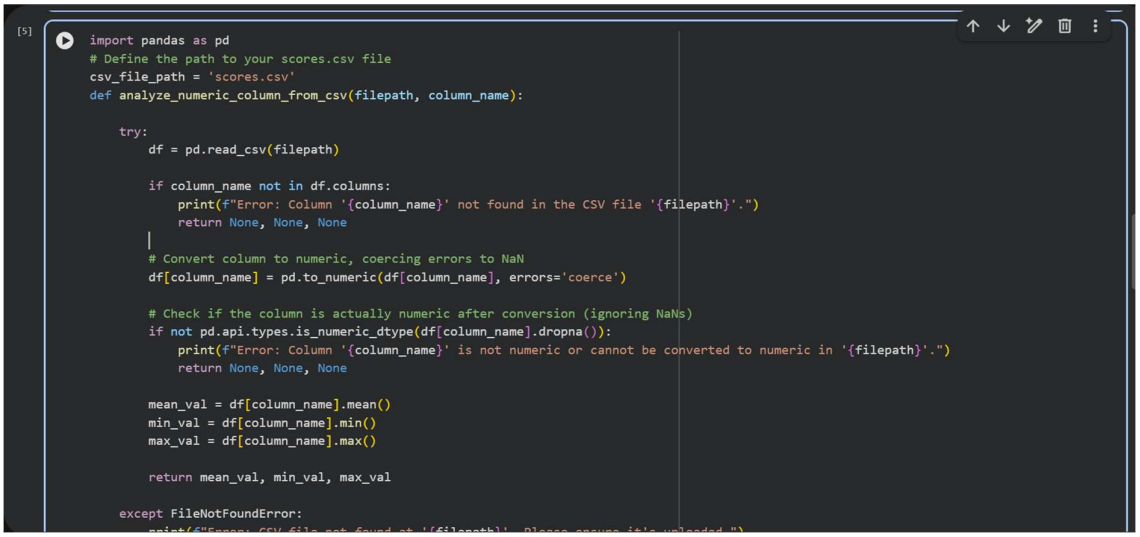
Scenario: You are processing structured data from a CSV file.
Task: Use Gemini in Colab to read a CSV file and calculate mean, min, and max.

Prompt:

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



Code Screenshot:



```
[5]
import pandas as pd
# Define the path to your scores.csv file
csv_file_path = 'scores.csv'
def analyze_numeric_column_from_csv(filepath, column_name):

    try:
        df = pd.read_csv(filepath)

        if column_name not in df.columns:
            print(f"Error: Column '{column_name}' not found in the CSV file '{filepath}'.")
            return None, None, None

        # Convert column to numeric, coercing errors to NaN
        df[column_name] = pd.to_numeric(df[column_name], errors='coerce')

        # Check if the column is actually numeric after conversion (ignoring NaNs)
        if not pd.api.types.is_numeric_dtype(df[column_name].dropna()):
            print(f"Error: Column '{column_name}' is not numeric or cannot be converted to numeric in '{filepath}'.")
            return None, None, None

        mean_val = df[column_name].mean()
        min_val = df[column_name].min()
        max_val = df[column_name].max()

        return mean_val, min_val, max_val

    except FileNotFoundError:
```

```
        except FileNotFoundError:
            print(f"Error: CSV file not found at '{filepath}'. Please ensure it's uploaded.")
            return None, None, None
        except Exception as e:
            print(f"An unexpected error occurred: {e}")
            return None, None, None

    # Specify the numeric column you want to analyze
    column_to_analyze = 'score' # Change this if your scores.csv has a different numeric column name

    # Call the function to analyze the specified column in scores.csv
    mean_value, min_value, max_value = analyze_numeric_column_from_csv(csv_file_path, column_to_analyze)

    if mean_value is not None:
        print(f"\nAnalysis for column '{column_to_analyze}' in '{csv_file_path}':")
        print(f"  Mean: {mean_value:.2f}")
        print(f"  Min: {min_value:.2f}")
        print(f"  Max: {max_value:.2f}")
```

OUTPUT(O/P):

```
...
    Analysis for column 'score' in 'scores.csv':
      Mean: 82.00
      Min: 67.00
      Max: 92.00
```

Explanation:

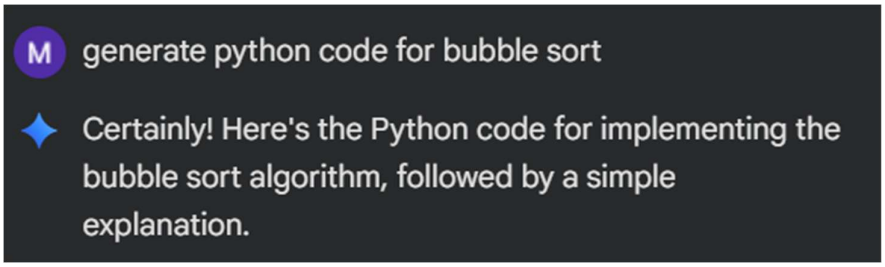
This program reads data from a CSV file using Python. It extracts numerical values from a column and calculates the mean, minimum, and maximum. CSV analysis is used in data processing and analytics applications.

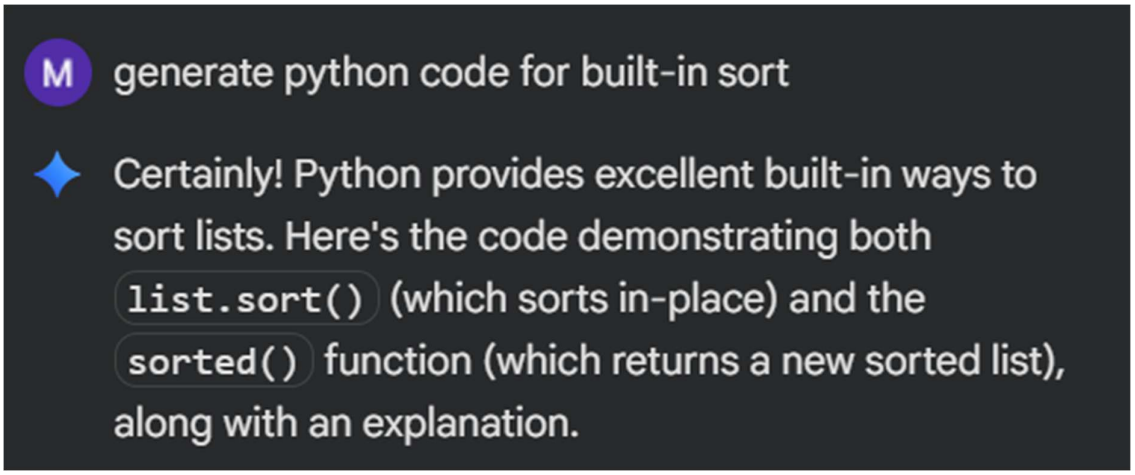
Task 4: Sorting Lists Manual vs Built-in

Scenario: You are reviewing algorithm choices for efficiency.

Task: Use Gemini to generate: Bubble sort Python's built-in sort()
Compare both implementations

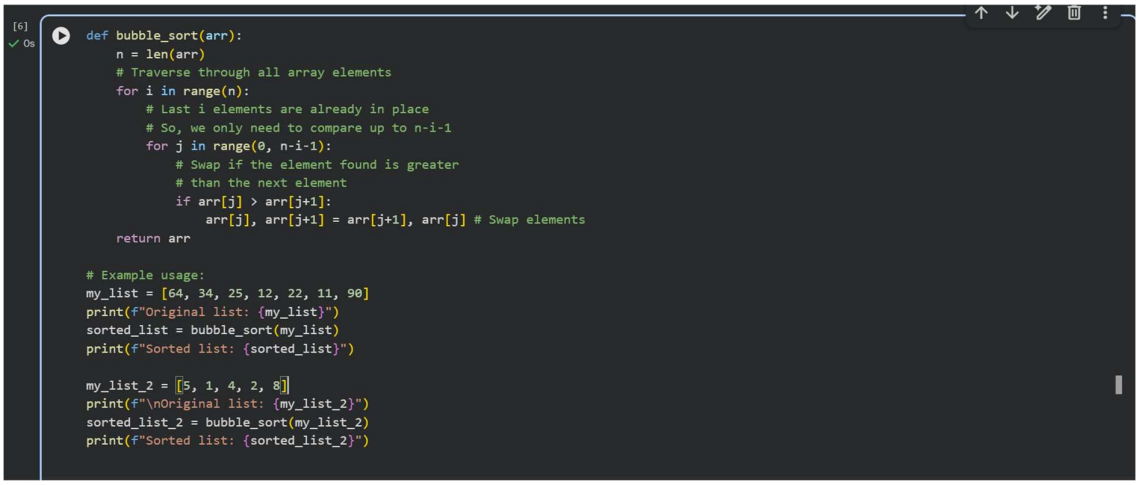
Prompt:





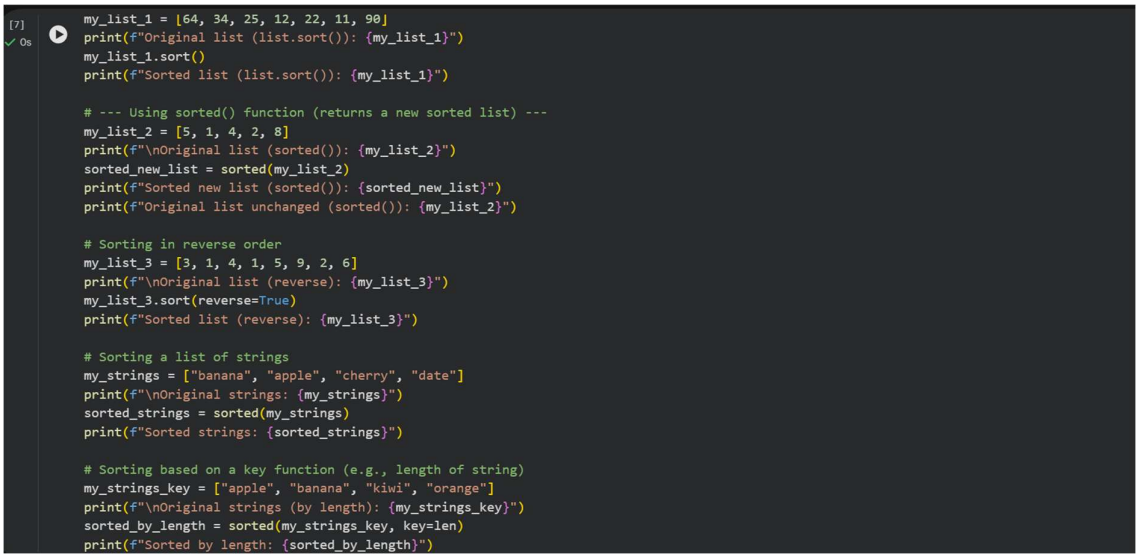
Code Screenshot:

Manual Bubble Sort code:



```
[6]
✓ 0s
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):
            # 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"\nOriginal list: {my_list_2}")
sorted_list_2 = bubble_sort(my_list_2)
print(f"Sorted list: {sorted_list_2}")
```

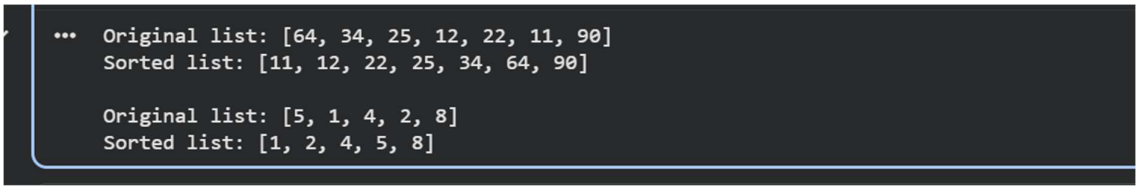
Sorting using sort() function:



```
[7]
✓ 0s
my_list_1 = [64, 34, 25, 12, 22, 11, 90]
print(f"Original list (list.sort()): {my_list_1}")
my_list_1.sort()
print(f"Sorted list (list.sort()): {my_list_1}")

# --- Using sorted() function (returns a new sorted list) ---
my_list_2 = [5, 1, 4, 2, 8]
print(f"\nOriginal list (sorted()): {my_list_2}")
sorted_new_list = sorted(my_list_2)
print(f"Sorted new list (sorted()): {sorted_new_list}")
print(f"Original list unchanged (sorted()): {my_list_2}")

# Sorting in reverse order
my_list_3 = [3, 1, 4, 1, 5, 9, 2, 6]
print(f"\nOriginal list (reverse): {my_list_3}")
my_list_3.sort(reverse=True)
print(f"Sorted list (reverse): {my_list_3}")

# Sorting a list of strings
my_strings = ["banana", "apple", "cherry", "date"]
print(f"\nOriginal strings: {my_strings}")
sorted_strings = sorted(my_strings)
print(f"Sorted strings: {sorted_strings}")

# Sorting based on a key function (e.g., length of string)
my_strings_key = ["apple", "banana", "kiwi", "orange"]
print(f"\nOriginal strings (by length): {my_strings_key}")
sorted_by_length = sorted(my_strings_key, key=len)
print(f"Sorted by length: {sorted_by_length}")
```

OUTPUT(O/P):

Bubble sort:



```
...  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]
```

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

Original list (reverse): [3, 1, 4, 1, 5, 9, 2, 6]
Sorted list (reverse): [9, 6, 5, 4, 3, 2, 1, 1]

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

Original strings (by length): ['apple', 'banana', 'kiwi', 'orange']
Sorted by length: ['kiwi', 'apple', 'banana', 'orange']
```

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.