2303A52366

Batch-31

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Assignment-2.3** | | | | |
|  | **Q.No.** | **Question** | ***Expected Time***  ***to complete*** |  |
|  | 1 | Lab 2: Exploring Additional AI Coding Tools beyond Copilot – Gemini (Colab) and Cursor AI  **Lab Objectives:**   * To explore and evaluate the functionality of Google Gemini for AI-assisted coding within Google Colab. * To understand and use Cursor AI for code generation, explanation, and refactoring. * To compare outputs and usability between Gemini, GitHub Copilot, and Cursor AI. * To perform code optimization and documentation using AI tools.   **Lab Outcomes (LOs):**  After completing this lab, students will be able to:   * Generate Python code using Google Gemini in Google Colab. * Analyze the effectiveness of code explanations and suggestions by Gemini. * Set up and use Cursor AI for AI-powered coding assistance. * Evaluate and refactor code using Cursor AI features. * Compare AI tool behavior and code quality across different platforms.   **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. * **Expected Output:**   + Prompts   #Write a Python code that reads a text file named 'data.txt' and counts the frequency of each word. The code should handle punctuation, ignore case sensitivity (treat 'Error' and 'error' as the same), and print the results in descending order of frequency.   * + Working code   import string  from collections import Counter  def count\_word\_frequency(file\_path):  try:  with open(file\_path, 'r', encoding='utf-8') as file:  # Read file and convert to lowercase  text = file.read().lower()    # Remove punctuation  text = text.translate(str.maketrans('', '', string.punctuation))    # Split into words  words = text.split()    # Count the frequency of each word  word\_counts = Counter(words)    # Print results sorted by highest frequency  print(f"{'Word':<15} | {'Frequency':<10}")  print("-" \* 28)  for word, count in word\_counts.most\_common():  print(f"{word:<15} | {count:<10}")    except FileNotFoundError:  print("Error: The file was not found.")  # Usage  count\_word\_frequency('data.txt')   * + Explanation   The script opens the file and converts everything to lowercase to ensure consistency.It uses "regular expressions" to strip away noise like commas, leaving only pure words.Finally, it uses a specialized counter to instantly rank these words by popularity, allowing us to identify the most frequent log entries without manual sorting.   * + Screenshot     **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 * **Expected Output:**   + Prompts   # write a python code to show  how to use the 'with' statement in  to perform basic I/O operations, including creating a text file, writing multiple lines, and reading the entire content back safely  # Using 'with' statement for file I/O operations in Python   * + Functional code   def main() -> None:  file\_name = 'sample.txt'    # Writing to the file  with open(file\_name, 'w') as file:  file.write("This is the first line of example text.\n")  file.write("This is the second line of example text.\n")  file.write("This is the third line of example text.\n")    # Reading from the file  with open(file\_name, 'r') as file:  content = file.read()    # Printing the content to the console  print("Content of 'sample.txt':")  print(content)  if \_\_name\_\_ == "\_\_main\_\_":  main()   * + Cursor AI screenshots     Explanation:This program utilizes the **with open()** context manager to handle file streams safely, ensuring the file is closed automatically after use. It first uses the **'w' mode** to create and populate 'sample.txt' with three lines of string data. Finally, it reopens the file in **'r' mode** to read the entire buffer and print it to the console, demonstrating a complete **input/output (I/O) cycle**.  **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:**   + Prompt   Write a Python code that read a CSV file and calculates the mean, min, and max of a specific numerical column.   * + Functional code   import csv  def calculate\_statistics(file\_name: str, column\_name: str) -> None:  values = []    # Reading the CSV file  with open(file\_name, mode='r') as csvfile:  csvreader = csv.DictReader(csvfile)  for row in csvreader:  try:  value = float(row[column\_name])  values.append(value)  except ValueError:  continue # Skip rows where conversion fails    if not values:  print(f"No valid data found in column '{column\_name}'.")  return    mean\_value = sum(values) / len(values)  min\_value = min(values)  max\_value = max(values)    print(f"Statistics for column '{column\_name}':")  print(f"Mean: {mean\_value}")  print(f"Min: {min\_value}")  print(f"Max: {max\_value}")  if \_\_name\_\_ == "\_\_main\_\_":  # Example usage - Using 'Age' column from data.csv  calculate\_statistics('data.csv', 'Age')   * + Correct output      * + Screenshot       Explanation:This script uses the **csv.DictReader** module to map each row to a dictionary, making it easy to access specific columns by name. It iterates through the file, converting target values to **floats** while handling potential errors like empty strings or text. Finally, it applies Python's built-in **sum()**, **min()**, and **max()** functions to calculate and display the statistical results.  **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:**   + Prompt   Write a Python code that implements the Bubble Sort algorithm and compares its execution time with built in sort method using a list of 5000 random integers.   * + Two versions of code   import time  import random  def bubble\_sort(arr):  n = len(arr)  # Manual Bubble Sort Implementation  for i in range(n):  for j in range(0, n - i - 1):  if arr[j] > arr[j + 1]:  arr[j], arr[j + 1] = arr[j + 1], arr[j]  return arr  def compare\_sorting\_methods():  # Generate a random list of integers  size = 1000  original\_data = [random.randint(0, 5000) for \_ in range(size)]    # 1. Bubble Sort (Manual)  data\_bubble = original\_data.copy()  start\_time = time.time()  bubble\_sort(data\_bubble)  bubble\_duration = time.time() - start\_time    # 2. Built-in sort() (Timsort)  data\_builtin = original\_data.copy()  start\_time = time.time()  data\_builtin.sort()  builtin\_duration = time.time() - start\_time    print(f"Sorting {size} elements:")  print(f"Manual Bubble Sort: {bubble\_duration:.6f} seconds")  print(f"Python Built-in sort(): {builtin\_duration:.6f} seconds")  print(f"\nConclusion: Python's built-in sort (Timsort) is significantly "  f"faster than manual Bubble Sort for large datasets.")  if \_\_name\_\_ == "\_\_main\_\_":  compare\_sorting\_methods()   * + Short comparison   Sorting 1000 elements:  Manual Bubble Sort: 0.126919 seconds  Python Built-in sort(): 0.000098 seconds  Conclusion: Python's built-in sort (Timsort) is significantly faster than manual Bubble Sort for large datasets.  **Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.** | Week1 - Monday |  |