AI Assistant Coding

Assignment-2.3

NAME :Hasini Irumalla

HT.NO: 2303A51286

BATCH :05

TASK-1: Word Frequency from Text File

PROMPT:

Analyzing log files for keyword frequency generate Python code that reads a text file and counts word frequency, then explains the code.

EXPLAINATION:

1. read_text_file(file_path)

- Opens the file with UTF-8 encoding
- Handles errors (file not found, read errors)
- Returns the file content as a string

2. clean_text(text)

- Uses regex re.sub(r'[^\w\s]', '', text) to remove punctuation
- Converts to lowercase so "Error" and "error" are counted together
- Returns cleaned text with only words and spaces

3. count_word_frequency(text)

- Splits text into words using split()
- Uses Counter from collections to count occurrences
- Returns a Counter object (dictionary-like with word counts)

4. display_results(word_frequency, top_n)

- Uses most_common(top_n) to get top N words by frequency
- Formats and prints results in a table
- Shows summary: total words and unique word count

5. analyze_log_file(file_path, top_n)

- Main function that orchestrates the process:

1. Reads file → 2. Cleans text → 3. Counts words → 4. Displays results

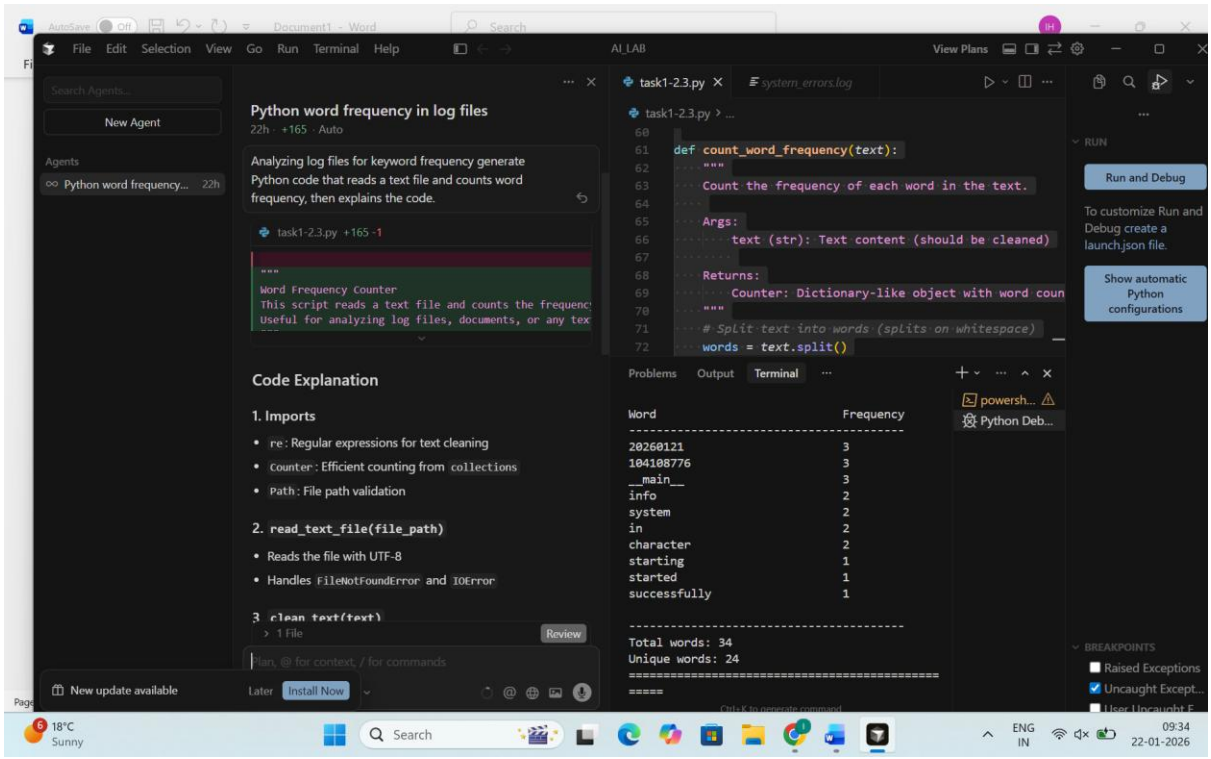
CODE:

```
import re
from collections import Counter
from pathlib import Path

def read_text_file(file_path):
    Args:
        file_path (str): Path to the text file
    Returns:
        str: Content of the file
    Raises:
        FileNotFoundError: If the file doesn't exist
        IOError: If there's an error reading the file
    try:
        with open(file_path, 'r', encoding='utf-8') as file:
            content = file.read()
        return content
    except FileNotFoundError:
        print(f"Error: File '{file_path}' not found.")
        raise
    except IOError as e:
        print(f"Error reading file: {e}")
        raise
def clean_text(text):
    Args:
        text (str): Raw text content
    Returns:
        str: Cleaned text with only alphanumeric characters and spaces
    cleaned = re.sub(r'[^\w\s]', '', text)
    cleaned = cleaned.lower()
    return cleaned
```

```
def count_word_frequency(text):
    Args:
        text (str): Text content (should be cleaned)
    Returns:
        Counter: Dictionary-like object with word counts
    words = text.split()
    word_frequency = Counter(words)
    return word_frequency
def display_results(word_frequency, top_n=None):
    Args:
        word_frequency (Counter): Word frequency counter object
        top_n (int, optional): Number of top words to display. If None, shows all.
    print("\n" + "="*20)
    print("WORD FREQUENCY ANALYSIS")
    print("="*20)
    sorted_words = word_frequency.most_common(top_n)
    if not sorted_words:
        print("No words found in the file.")
        return
    print(f"\n{'Word':<30} {'Frequency':<10}")
    print("-" * 40)
    for word, count in sorted_words:
        print(f"{word:<30} {count:<10}")
    total_words = sum(word_frequency.values())
    unique_words = len(word_frequency)
    print("\n" + "-" * 40)
    print(f"Total words: {total_words}")
    print(f"Unique words: {unique_words}")
    print("="*50 + "\n")
def analyze_log_file(file_path, top_n=10):
```

```
    Main function to analyze a log file for keyword frequency.
    Args:
        file_path (str): Path to the text/log file
        top_n (int): Number of top words to display (default: 10)
    print(f"Analyzing file: {file_path}")
    raw_text = read_text_file(file_path)    cleaned_text =
clean_text(raw_text)    word_frequency = count_word_frequency(cleaned_text)
         display_results(word_frequency, top_n)
    return word_frequency
if __name__ == "__main__":
    log_file = "system_errors.log
    if Path(log_file).exists():
        analyze_log_file(log_file, top_n=10)
    else:
        print(f"File '{log_file}' not found in current directory.")
        print("\nTo use this script with your own file:")
        print("1. Update the 'log_file' variable with your file path")
        print("2. Or modify the function call: analyze_log_file('your_file.txt')")
        user_file = input("\nEnter file path (or press Enter to skip): ").strip()
        if user_file:
            try:
                analyze_log_file(user_file, top_n=10)
            except Exception as e:
                print(f"Error: {e}")
```

OUTPUT:



TASK-2: File Operations Using Cursor AI

PROMPT :

automating basic file operations.Creates a text fileWrites sample text,Reads and displays the content.Generate a python code

EXPLAINATION:

What it does: Creates a text file, writes sample text, then reads and displays it.

Functions:

1. **create_and_write_file()** — Opens a file in write mode, writes content, closes the file.
2. **read_and_display_file()** — Opens a file in read mode, reads content, prints it with separators. Handles errors.
3. **main()** — Sets filename and sample text, calls the write function, then the read function.

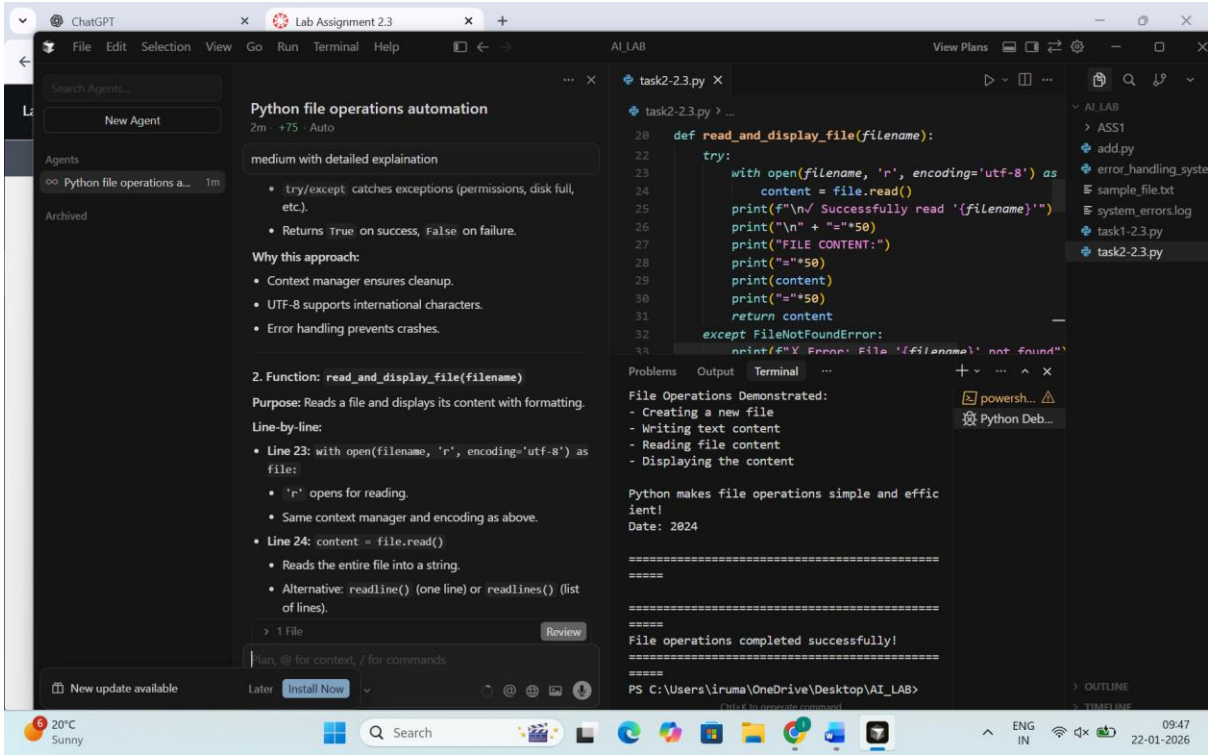
Flow: Run → main() → create file → write text → read file → display content.

CODE:

```
def create_and_write_file(filename, content):
    try:
        with open(filename, 'w', encoding='utf-8') as file:
            file.write(content)
        print(f"✓ Successfully created and wrote to '{filename}'")
        return True
    except Exception as e:
        print(f"✗ Error writing to file: {e}")
        return False
def read_and_display_file(filename):
    try:
        with open(filename, 'r', encoding='utf-8') as file:
            content = file.read()
        print(f"\n✓ Successfully read '{filename}'")
        print("\n" + "="*50)
        print("FILE CONTENT:")
        print("="*50)
        print(content)
        print("="*50)
        return content
    except FileNotFoundError:
        print(f"✗ Error: File '{filename}' not found")
        return None
    except Exception as e:
        print(f"✗ Error reading file: {e}")
        return None
def main():
```

```
filename = "sample_file.txt"
    sample_text = """Hello, World!
    print("="*50)
    print("AUTOMATING BASIC FILE OPERATIONS")
    print("="*50)
    print("\n[Step 1] Creating file and writing sample text...")
    if create_and_write_file(filename, sample_text):
        print("\n[Step 2] Reading and displaying file content...")
        read_and_display_file(filename)
        print("\n" + "="*50)
        print("File operations completed successfully!")
        print("="*50)
    else:
        print("\nFailed to complete file operations.")
if __name__ == "__main__":
    main()
```

OUTPUT:



TASK-3 :C SV Data Analysis

PROMPT :

structured data from a CSV file.read a CSV file and calculate mean, min, and max.Generate a python code

EXPLAINATION :

What it does: Reads a CSV and computes mean, min, and max for numeric columns.

Key Functions:

1. **read_csv_file()** - Opens CSV, reads rows as dictionaries
2. **convert_to_numeric()** - Converts strings to numbers (returns None if invalid)
3. **calculate_statistics()** - For each column:

- Extracts values → converts to numbers → filters out None
- Calculates mean, min, max, count (only for numeric columns)

4. **display_statistics()** - Prints formatted results
5. **main()** - Runs: read → calculate → display

How it works:

- Uses csv.DictReader to read CSV rows as dictionaries
- Automatically detects numeric columns (skips text columns like "Name")
- Uses statistics.mean() for mean, min()/max() for min/max
- Handles errors gracefully (missing files, invalid data)

Result: Shows mean, min, max, and count for each numeric column in the CSV.

CODE :

```
import csv
import statistics
from pathlib import Path
from typing import Dict, List, Any
def read_csv_file(file_path: str) -> List[Dict[str, Any]]:
    Args:
        file_path (str): Path to the CSV file
    Returns:
        List[Dict[str, Any]]:
```

```
        try:
            data = []
            with open(file_path, 'r', encoding='utf-8') as file:
                csv_reader = csv.DictReader(file)
                for row in csv_reader:
                    data.append(row)
            return data
        except FileNotFoundError:
            print(f"Error: File '{file_path}' not found.")
            raise
        except IOError as e:
            print(f"Error reading file: {e}")
            raise
def identify_numeric_columns(data: List[Dict[str, Any]]) -> List[str]:
    Args:
        data (List[Dict[str, Any]]): CSV data as list of dictionaries
    Returns:
        List[str]:

    if not data:
        return []
    numeric_columns = []
    column_names = data[0].keys()
    for column in column_names:
        numeric_values = []
        for row in data:
            value = row[column].strip() if row[column] else None
            if value:
                try:
                    float_value = float(value)
```

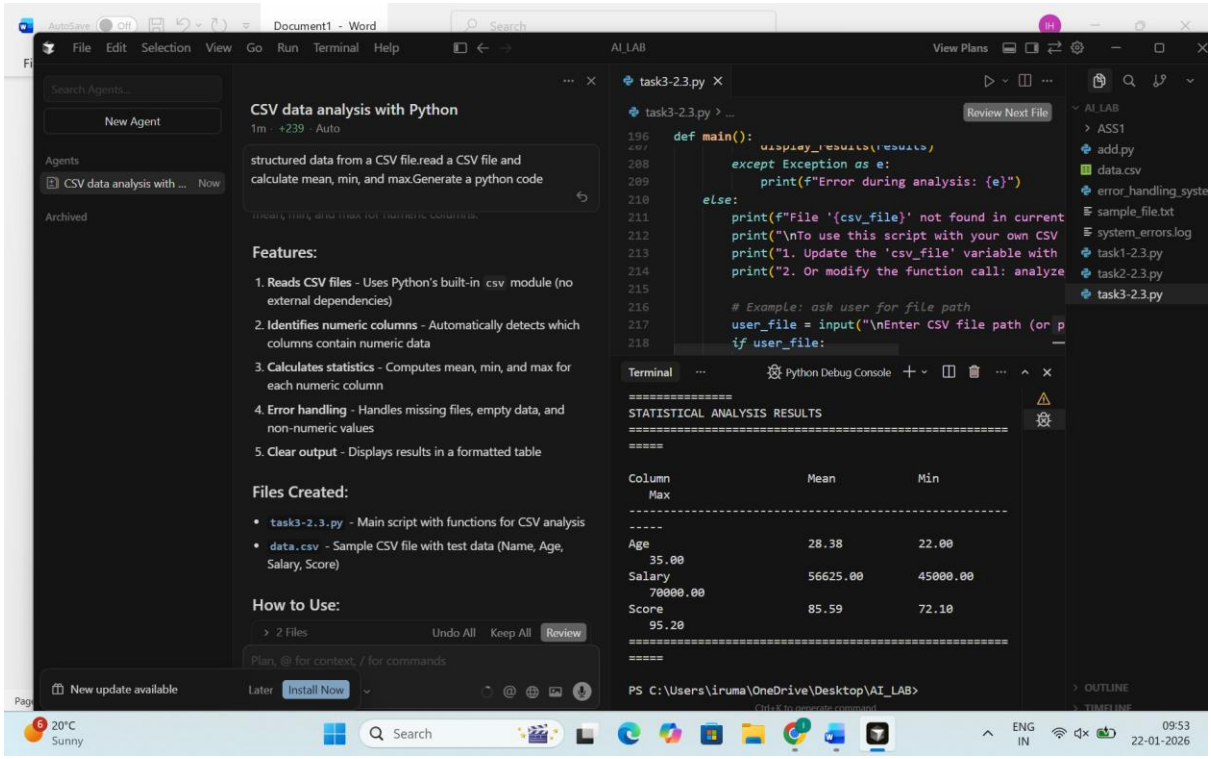
```
                numeric_values.append(float_value)
            except ValueError:
                break
            if len(numeric_values) == len([r for r in data if r[column].strip()]):
                numeric_columns.append(column)
    return numeric_columns
def extract_numeric_values(data: List[Dict[str, Any]], column: str) -> List[float]:    Args:
    data (List[Dict[str, Any]]): CSV data as list of dictionaries
    column (str): Column name to extract values from

    Returns:
    List[float]:
    values = []
    for row in data:
        value = row[column].strip() if row[column] else None
        if value:
            try:
                values.append(float(value))
            except ValueError:
                        continue
    return values
def calculate_statistics(values: List[float]) -> Dict[str, float]:
    Args:
    values (List[float]):
    Returns:
    Dict[str, float]:
    if not values:
        return {"mean": None, "min": None, "max": None}
    return {
        "mean": statistics.mean(values),
```

```
        "min": min(values),
        "max": max(values)
    }

def analyze_csv_file(file_path: str) -> Dict[str, Dict[str, float]]:
    Args:
        file_path (str): Path to the CSV file    Returns:
        Dict[str, Dict[str, float]]:
    print(f"Analyzing CSV file: {file_path}")
    print("="*60)
    data = read_csv_file(file_path)
    if not data:
        print("Error: CSV file is empty or contains no data.")
        return {}
    print(f"✓ Successfully read {len(data)} rows from CSV file")
    numeric_columns = identify_numeric_columns(data)    if not numeric_columns:
        print("Warning: No numeric columns found in the CSV file.")
        return {}
    print(f"✓ Found {len(numeric_columns)} numeric column(s): {',
'.join(numeric_columns)}")
    results = {}
    for column in numeric_columns:
        values = extract_numeric_values(data, column)
        if values:
            stats = calculate_statistics(values)
            results[column] = stats

    return results
def display_results(results: Dict[str, Dict[str, float]]):
    Args:
        results (Dict[str, Dict[str, float]]): Dictionary containing statistics
```

```
    if not results:
        print("\nNo statistics to display.")
        return
    print("\n" + "="*60)
    print("STATISTICAL ANALYSIS RESULTS")
    print("="*60)
    print(f"\n{'Column':<25} {'Mean':<15} {'Min':<15} {'Max':<15}")
    print("-" * 60)    for column, stats in results.items():
        mean_val = f"{stats['mean']:.2f}" if stats['mean'] is not None else "N/A"
        min_val = f"{stats['min']:.2f}" if stats['min'] is not None else "N/A"
        max_val = f"{stats['max']:.2f}" if stats['max'] is not None else
"N/A"        print(f"{column:<25} {mean_val:<15} {min_val:<15} {max_val:<15}")
    print("="*60 + "\n")
def main():
    csv_file = "data.csv"
    if Path(csv_file).exists():
        try:
            results = analyze_csv_file(csv_file)
            display_results(results)
        except Exception as e:
            print(f"Error during analysis: {e}")
    else:
        print(f"File '{csv_file}' not found in current directory.")
        print("\nTo use this script with your own CSV file:")
        print("1. Update the 'csv_file' variable with your file path")
        print("2. Or modify the function call: analyze_csv_file('your_file.csv')")
            user_file = input("\nEnter CSV file path (or press Enter to skip): ").strip()
        if user_file:
            try:
                results = analyze_csv_file(user_file)
                display_results(results)
```

```
        except Exception as e:
            print(f"Error: {e}")
if __name__ == "__main__":
    main()
```

OUTPUT:



TASK-4 : Sorting Lists – Manual vs Built-in

PROMPT :

reviewing algorithm choices for efficiency.

to generate:

Bubble sort,Python’s built-in sort(),Compare both implementations.Generate using python code

EXPLAINATION :

Purpose: Compares Bubble Sort with Python's built-in sort().

Main Components:

1. **bubble_sort() (lines 6-38)**

- Custom bubble sort: compares adjacent elements and swaps if out of order

- Time: O(n²); Space: O(1)
- Early exit if no swaps occur

2. **python_builtin_sort() (lines 41-54)**

- Uses Python's built-in sort() (Timsort)
- Time: O(n log n); Space: O(n)

3. **compare_sorting_algorithms() (lines 57-128)**

- Runs both algorithms on the same array
- Measures execution time (averaged over multiple runs)
- Verifies both produce identical results
- Reports performance comparison

4. **main() (lines 131-163)**

- Tests with different array sizes (100, 500, 1000)
- Tests random, sorted, and reverse-sorted arrays
- Calls the comparison function for each test case

Key Takeaway: The script demonstrates that Python's built-in sort() is faster than bubble sort, especially on larger arrays, due to its better time complexity (O(n log n) vs O(n²)).

CODE:

```
import time
import random
from typing import List, Tuple
def bubble_sort(arr: List[int]) -> List[int]:
    Args:
        arr (List[int]): List of integers to sort
    Returns:
        List[int]:
    arr = arr.copy()
    n = len(arr)
    for i in range(n):
        swapped = False
```

```
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True        if not swapped:
            break    return arr
def python_builtin_sort(arr: List[int]) -> List[int]:
    Args:
        arr (List[int]): List of integers to sort
    Returns:
        List[int]:
    arr = arr.copy()
    arr.sort()  # In-place sort using Timsort
    return arr
def measure_execution_time(func, arr: List[int]) -> Tuple[List[int], float]:


      Args:
        func: Sorting function to measure
        arr: List to sort    Returns:
        Tuple containing (sorted_list, execution_time_in_seconds)


    start_time = time.perf_counter()
    sorted_arr = func(arr)
    end_time = time.perf_counter()
    execution_time = end_time - start_time
    return sorted_arr, execution_time
def generate_test_data(size: int, data_type: str = "random") -> List[int]:
    Args:
        size: Number of elements in the array
        data_type: Type of data to generate
            - "random": Random integers
```

```
        - "sorted": Already sorted array
        - "reversed": Reverse sorted array
        - "nearly_sorted": Mostly sorted with few swaps
    Returns:
        List[int]:
    if data_type == "random":
        return [random.randint(1, 10000) for _ in range(size)]
    elif data_type == "sorted":
        return list(range(1, size + 1))
    elif data_type == "reversed":
        return list(range(size, 0, -1))
    elif data_type == "nearly_sorted":
        arr = list(range(1, size + 1))
        # Swap 5% of elements randomly
        swap_count = max(1, size // 20)
        for _ in range(swap_count):
            i = random.randint(0, size - 1)
            j = random.randint(0, size - 1)
            arr[i], arr[j] = arr[j], arr[i]
        return arr
    else:
        return [random.randint(1, 10000) for _ in range(size)]
def verify_correctness(original: List[int], sorted_arr: List[int]) -> bool:
    Args:
        original:
        sorted_arr :
    Returns:
    if len(original) != len(sorted_arr):
        return False
    for i in range(len(sorted_arr) - 1):
```

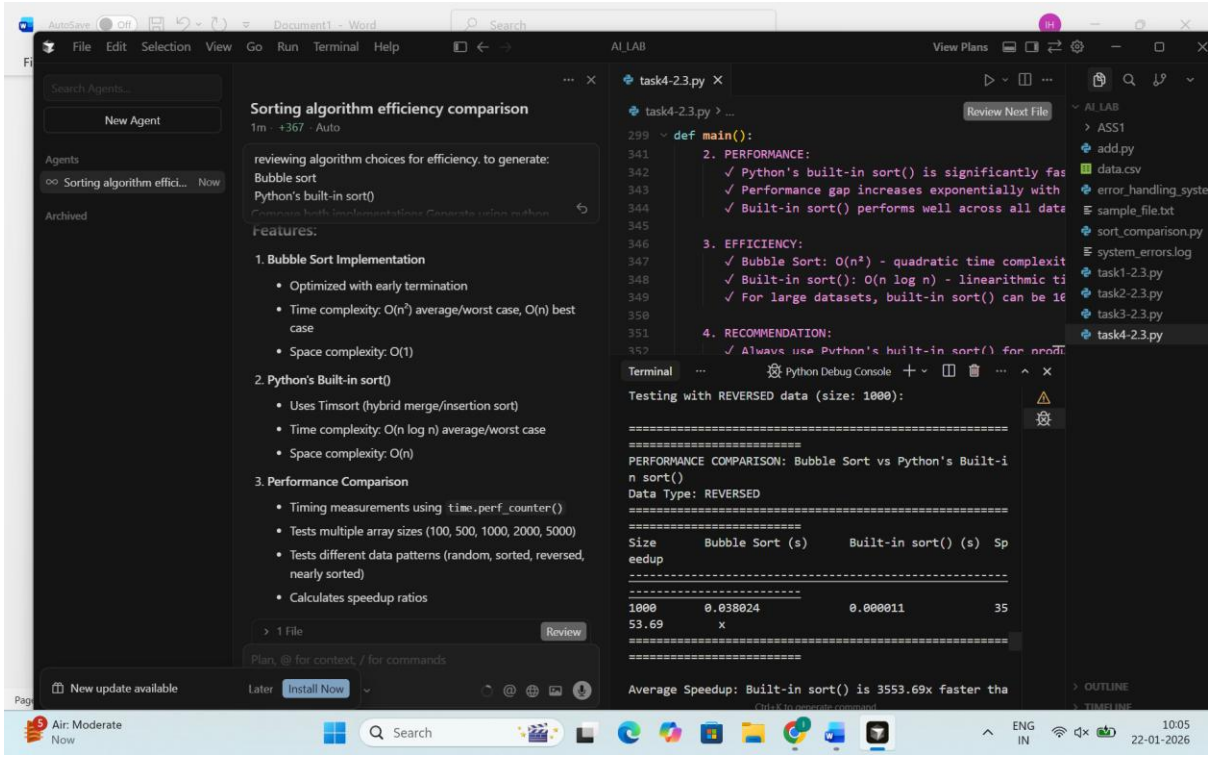
```
        if sorted_arr[i] > sorted_arr[i + 1]:
            return False
    return sorted(original) == sorted_arr
def compare_performance(sizes: List[int], data_type: str = "random",
                        num_trials: int = 3) -> None:
    Args:
        sizes: List of array sizes to test
        data_type:
        num_trials    print("\n" + "="*80)
    print(f"PERFORMANCE COMPARISON: Bubble Sort vs Python's Built-in sort()")
    print(f"Data Type: {data_type.upper()}")
    print("="*80)
    print(f"{'Size':<10} {'Bubble Sort (s)':<20} {'Built-in sort() (s)':<20} {'Speedup':<15}")
    print("-"*80)
    bubble_times = []
    builtin_times = []
      for size in sizes:
        bubble_total = 0
        builtin_total = 0
        for trial in range(num_trials):
            test_data = generate_test_data(size, data_type)
            _, bubble_time = measure_execution_time(bubble_sort, test_data)
            bubble_total += bubble_time            _, builtin_time =
measure_execution_time(python_builtin_sort, test_data)
            builtin_total += builtin_time
        bubble_avg = bubble_total / num_trials
        builtin_avg = builtin_total / num_trials
        speedup = bubble_avg / builtin_avg if builtin_avg > 0 else float('inf')
        bubble_times.append(bubble_avg)
        builtin_times.append(builtin_avg)
```

```
            print(f"{size:<10} {bubble_avg:<20.6f} {builtin_avg:<20.6f}
{speedup:<15.2f}x")    print("="*80)
    speedups = [b / bi for b, bi in zip(bubble_times, builtin_times) if bi > 0]
    if speedups:
        avg_speedup = sum(speedups) / len(speedups)
        print(f"\nAverage Speedup: Built-in sort() is {avg_speedup:.2f}x faster than Bubble
Sort")
    print()
def demonstrate_correctness():    print("\n" + "="*80)
    print("CORRECTNESS VERIFICATION")
    print("="*80)    test_array = [64, 34, 25, 12, 22, 11, 90, 5]
    print(f"\nOriginal array: {test_array}")    bubble_result = bubble_sort(test_array)
    print(f"Bubble sort result: {bubble_result}")    builtin_result =
python_builtin_sort(test_array)
    print(f"Built-in sort() result: {builtin_result}")
    bubble_correct = verify_correctness(test_array, bubble_result)
    builtin_correct = verify_correctness(test_array, builtin_result)
        print(f"\n✓ Bubble sort correctness: {bubble_correct}")
    print(f"✓ Built-in sort() correctness: {builtin_correct}")
    print(f"✓ Both produce identical results: {bubble_result == builtin_result}")
    print("="*80)
def theoretical_analysis():
    print("\n" + "="*80)
    print("THEORETICAL COMPLEXITY ANALYSIS")
    print("="*80)
    print("\n1. BUBBLE SORT:")
    print("   Time Complexity:")
    print("     • Best Case:    O(n)   - when array is already sorted (with optimization)")
    print("     • Average Case: O(n²)  - typical random data")
    print("     • Worst Case:   O(n²)  - when array is in reverse order")
    print("   Space Complexity: O(1)   - in-place sorting, constant extra space")
```

```
print("   Stability:        Stable - maintains relative order of equal elements")
print("   Adaptive:         Yes    - can detect early termination")
print("   Algorithm Type:   Comparison-based, simple but inefficient")
  print("\n2. PYTHON'S BUILT-IN sort() (Timsort):")
print("   Time Complexity:")
print("     • Best Case:    O(n)   - when array is already sorted")
print("     • Average Case: O(n log n) - typical random data")
print("     • Worst Case:   O(n log n) - guaranteed")
print("   Space Complexity: O(n)   - may use additional space")
print("   Stability:        Stable - maintains relative order of equal elements")
print("   Adaptive:         Yes    - optimized for real-world data patterns")
print("   Algorithm Type:   Hybrid (Merge Sort + Insertion Sort)")
print("   Notes: Timsort is designed to perform well on many")
print(kinds of real-world data, including partially")
print("sorted arrays")
print("\n3. EFFICIENCY COMPARISON:")
print("   • For small arrays (< 10 elements): Both are fast, difference is negligible")
print("   • For medium arrays (100-1000 elements): Built-in sort() is 10-100x faster")
print("   • For large arrays (> 1000 elements): Built-in sort() is 100-1000x faster")
print("   • As data size increases, the performance gap grows exponentially")
  print("\n4. WHEN TO USE EACH:")
print("   Bubble Sort:")
print("     • Educational purposes only")
print("     • Very small datasets (< 10 elements)")
print("     • When simplicity is more important than performance")
print("     • NOT recommended for production code")
print("   Python's Built-in sort():")
print("     • Always prefer for production code")
print("     • Handles all data sizes efficiently")
print("     • Optimized for real-world scenarios")
```

```
    print("     • Well-tested, reliable, and maintained")
      print("\n5. KEY TAKEAWAYS:")
    print("   • Bubble sort: O(n²) complexity makes it inefficient for large datasets")
    print("   • Built-in sort(): O(n log n) complexity scales much better")
    print("   • The performance difference increases dramatically with data size")
    print("   • Always use built-in sort() for real applications")
    print("="*80 + "\n")
def main():
    print("\n" + "="*80)
    print("ALGORITHM EFFICIENCY COMPARISON")
    print("Bubble Sort vs Python's Built-in sort()")
    print("="*80)
       theoretical_analysis()
    demonstrate_correctness()    print("\n" + "="*80)
    print("PERFORMANCE TESTING WITH DIFFERENT ARRAY SIZES")
    print("="*80)
    test_sizes = [100, 500, 1000, 2000, 5000]
    compare_performance(test_sizes, data_type="random", num_trials=3)
    print("\n" + "="*80)
    print("PERFORMANCE WITH DIFFERENT DATA PATTERNS")
    print("="*80)
    for data_type in ["random", "sorted", "reversed", "nearly_sorted"]:
       print(f"\nTesting with {data_type.upper()} data (size: 1000):")
       compare_performance([1000], data_type=data_type, num_trials=5)
    print("\n" + "="*80)
    print("SUMMARY AND CONCLUSION")
    print("="*80)
    print("="*80 + "\n")
if __name__ == "__main__":
    main()
```

OUTPUT :


Sorting algorithm efficiency comparison
reviewing algorithm choices for efficiency. to generate:
Bubble sort
Python's built-in sort()
Features:
1. Bubble Sort Implementation
2. Python's Built-in sort()
3. Performance Comparison
task4-2.3.py
def main():
2. PERFORMANCE:
3. EFFICIENCY:
4. RECOMMENDATION:
Terminal
Python Debug Console
Testing with REVERSED data (size: 1000):
PERFORMANCE COMPARISON: Bubble Sort vs Python's Built-in sort()
Data Type: REVERSED
Size Bubble Sort (s) Built-in sort() (s) Speedup
1000 0.038024 0.000011 3553.69x
Average Speedup: Built-in sort() is 3553.69x faster tha