

## AIAC-2

Name: Siri Vempati

Ht.no:2303A51309

Batch no: 05

Task-1: Word Frequency from Text File

Prompt:

Generate Python code that reads a text file and counts word frequency

Code:

```
from collections import Counter

import re

import sys

from pathlib import Path

def word_frequencies(file_path: str) -> Counter:
    text = Path(file_path).read_text(encoding="utf-8", errors="ignore")

    # Extract words (letters/digits/underscore). Lowercase for case-insensitive
    # counting.

    words = re.findall(r"\b\w+\b", text.lower())
    return Counter(words)

def _prompt_for_path_and_top_n() -> tuple[str, int]:
    path = input("Enter path to text file (e.g. sample_clean.txt): ").strip().strip('"')

    if not path:
        if Path("sample_clean.txt").exists():
            path = "sample_clean.txt"
        elif Path("sample.txt").exists():


```

```
path = "sample.txt"

else:
    raise SystemExit("No file path provided.")

top_raw = input("How many top words to show? (default 05sa): ").strip()
top_n = int(top_raw) if top_raw else 5

return path, top_n

def main() -> None:
    if len(sys.argv) < 2:
        # Debug Console / no-args mode
        path, top_n = _prompt_for_path_and_top_n()

    else:
        path = sys.argv[1]
        top_n = int(sys.argv[2]) if len(sys.argv) >= 3 else 20

    counts = word_frequencies(path)
    print(f"Total unique words: {len(counts)}")
    print(f"Top {top_n} words:")
    for word, freq in counts.most_common(top_n):
        print(f"{word}: {freq}")

if __name__ == "__main__":
    main()

Output:
```

The screenshot shows the AI Assistant Coding interface. On the left, there's a sidebar with 'Agents' and a search bar. The main area has tabs for 'Python word frequency counter' and 'Review: Python word frequency counter'. The code editor shows a file named 'word\_count.py' with the following content:

```

from collections import Counter
import re
import sys
from pathlib import Path

def word_frequencies(file_path: str) -> Counter:
    """
    Read a text file and return word-frequency counts (case-insensitive).
    Words are extracted using a regex so punctuation is ignored.
    """
    # Read the file the user provided (not a hardcoded filename)
    text = Path(file_path).read_text(encoding="utf-8", errors="ignore")

    # Extract words (letters/digits/underscore), Lowercase for case-insensitive counting
    words = re.findall(r"\b\w+\b", text.lower())
    return Counter(words)

```

Below the code editor is a terminal window showing the execution of the program:

```

PS C:\Users\vempa\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING> python word_count.py sample_clean.txt
Enter path to text file (e.g., sample_clean.txt): sample.txt
How many top words to show? (default 0): 5
Total unique words: 13
Top 5 words:
hello: 3
world: 2
hcl: 2
nhello: 1
n: 1

```

## Explanation:

- The program **reads a text file** you give it.
- It **converts all text to lowercase** and **extracts words** (ignoring punctuation).
- It uses **Counter** to **count each word's frequency**.
- It then **prints the most common words** (Top N).
- If you run it in **Debug** without arguments
- It **asks for the file name and Top N** in the console.

## Task-02- File Operations Using Cursor AI

**Prompt:** Generate a text file and writes sample text, reads and displays the content.

### Code:

```
def create_and_write_file(filename: str, content: str) -> None:
```

```
    with open(filename, 'w', encoding='utf-8') as file:
```

```
        file.write(content)
```

```
    print(f"Successfully created and wrote to '{filename}'")
```

```
def read_and_display_file(filename: str) -> None
try:
    with open(filename, 'r', encoding='utf-8') as file:
        content = file.read()
        print(f"\n{'='*60}")
        print(f"Content of '{filename}':")
        print(f"{'='*60}")
        print(content)
        print(f"{'='*60}\n")
except FileNotFoundError:
    print(f"Error: File '{filename}' not found.")
except Exception as e:
    print(f"Error reading file: {e}")
def main():
    # Define the filename
    filename = "sample_output.txt"
    # Sample text content
    sample_text = """Hello, World!
```

This is a sample text file created by Python.

It demonstrates file operations including:

- Creating a new file
- Writing text content
- Reading the file back
- Displaying the content

Python makes file handling simple and efficient.

You can use this as a template for your own file operations.

Have a great day!!!!

*# Step 1: Generate and write to the text file*

```
print("Step 1: Creating text file and writing sample text...")  
create_and_write_file(filename, sample_text)
```

*# Step 2: Read and display the content*

```
print("Step 2: Reading and displaying file content...")  
read_and_display_file(filename)
```

```
if __name__ == "__main__":  
    main()
```

## Output:

The screenshot shows the AI Assistant Coding interface. On the left, there's a sidebar with 'Search Agents...', 'New Agent', and a list of agents: 'Text file generation an...' and 'Python word frequenc...'. The main area has tabs for 'Text file generation and display' and 'Brief Code Explanation'. The 'Text file generation and display' tab contains the Python code. The 'Brief Code Explanation' tab provides a brief explanation of what the code does, lists three functions (create\_and\_write\_file, read\_and\_display\_file, main), and details the flow of execution. On the right, the terminal window shows the execution of the code, creating a file named 'sample.txt' with the content 'Hello, World!', and then reading it back. The bottom status bar shows '1 File' and other system information.

## Explanation:

**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.

### **Task-03**- CSV Data Analysis

**Prompt:** To read a CSV file and calculate mean, min, and max.

#### **Code:**

```
import csv
import statistics
from typing import Dict, List, Any

def read_csv_file(filename: str) -> List[Dict[str, Any]]:
    data = []
    try:
        with open(filename, 'r', encoding='utf-8') as file:
            csv_reader = csv.DictReader(file)
            for row in csv_reader:
                data.append(row)
        print(f"Successfully read {len(data)} rows from '{filename}'")
    return data

except FileNotFoundError:
    print(f"Error: File '{filename}' not found.")
    return []

except Exception as e:
```

```
print(f"Error reading CSV file: {e}")

return []

def convert_to_numeric(value: str) -> float:
    try:
        return float(value)
    except (ValueError, TypeError):
        return None

def calculate_statistics(data: List[Dict[str, Any]]) -> Dict[str, Dict[str, float]]:
    if not data:
        print("No data to process.")

    return {}

    # Get all column names from the first row
    columns = list(data[0].keys())

    # Dictionary to store statistics for each numeric column
    stats = {}

    for column in columns:
        # Extract values for this column
        values = [row[column] for row in data]

        # Convert to numeric values, filtering out None values
        numeric_values = [convert_to_numeric(val) for val in values]
        numeric_values = [val for val in numeric_values if val is not None]

        # Only calculate statistics if we have numeric values
        if numeric_values:
            stats[column] = {
                'mean': statistics.mean(numeric_values),
                'min': min(numeric_values),
```

```
'max': max(numeric_values),  
    'count': len(numeric_values)  
  
}  
  
return stats  
  
def display_statistics(stats: Dict[str, Dict[str, float]]) -> None:  
    """Display the calculated statistics in a formatted way."""  
  
if not stats:  
  
    print("No numeric columns found in the CSV file.")  
  
return  
  
print("\n" + "="*70)  
  
print("STATISTICS SUMMARY")  
  
print("=*70)  
  
for column, values in stats.items():  
  
    print(f"\nColumn: {column}")  
  
    print(f" Count: {values['count']}")  
  
    print(f" Mean: {values['mean']:.2f}")  
  
    print(f" Min: {values['min']:.2f}")  
  
    print(f" Max: {values['max']:.2f}")  
  
print("=*70 + "\n")  
  
def main():  
  
    # CSV filename - change this to your CSV file name  
  
    csv_filename = "data.csv"  
  
print(f"Reading CSV file: {csv_filename}")  
  
print("-" * 70)  
  
# Step 1: Read the CSV file  
  
data = read_csv_file(csv_filename)
```

```
if not data:  
    print("\nNo data was read. Please check if the file exists and is valid.")  
    return  
  
# Step 2: Calculate statistics  
  
print("\nCalculating statistics...")  
  
stats = calculate_statistics(data)  
  
# Step 3: Display the results  
  
display_statistics(stats)  
  
if __name__ == "__main__":  
    main()
```

## **Output:**

Calculating statistics...

STATISTICS SUMMARY

Column: Age

Count: 8

Mean: 28.38

Min: 22.00

Max: 35.00

Column: Salary

Count: 8

Mean: 56500.00

Min: 45000.00

Max: 70000.00

Column: Score

Count: 8

Mean: 87.59

Min: 78.90

Max: 95.20

The screenshot shows the AI Assistant Coding interface. On the left, there's a sidebar with 'Search Agents...', 'New Agent', and a list of agents: 'CSV file statistics calcu...', 'Text file generation an...', and 'Python word frequenc...'. The main area has tabs for 'AI ASSISTANT CODING', 'word\_count(T1).py', 'text\_file\_operations.py', 'csv\_statistics.py' (which is the active tab), and 'sample.txt'. The 'csv\_statistics.py' tab contains Python code for reading CSV files and calculating statistics. The code includes functions for reading CSV files, converting strings to numeric values, calculating statistics for numeric columns, and displaying results. The terminal below shows the output of running the script on a CSV file, including a statistics summary table.

```
def read_csv_file(filename: str) -> List[Dict[str, Any]]:
    print(f"Error reading CSV file: {e}")
    return []

def convert_to_numeric(value: str) -> float:
    """Convert a string value to float, handling errors gracefully."""
    try:
        return float(value)
    except (ValueError, TypeError):
        return None

def calculate_statistics(data: List[Dict[str, Any]]) -> Dict[str, Dict[str, float]]:
    """Calculate mean, min, and max for all numeric columns in the CSV."""
    if not data:
        print("No data to process.")
        return {}

Column: Age
Count: 8
Mean: 28.38
Min: 22.00
Max: 35.00

Column: Salary
Count: 8
```

## Explanation:

**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.

## Task-04: Sorting Lists – Manual vs Built-in

**Prompt:** To generate:1)Bubble sort,2) Python's built-in sort(),3)Compare both implementations.

### Code:

```
import time  
import random  
from typing import List  
  
def bubble_sort(arr: List[int]) -> List[int]:  
    """  
    Implement bubble sort algorithm.  
    """
```

Bubble sort repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Time Complexity:  $O(n^2)$

Space Complexity:  $O(1)$

"""

```
# Create a copy to avoid modifying the original list
```

```

arr = arr.copy()
n = len(arr)

# Traverse through all array elements
for i in range(n):

    # Flag to optimize: if no swaps occur, array is already sorted
    swapped = False

    # 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]

            swapped = True

    # If no two elements were swapped by inner loop, then break
    if not swapped:
        break

return arr

```

```
def python_builtin_sort(arr: List[int]) -> List[int]:
```

```
    ....
```

Use Python's built-in `sort()` method.

Python's built-in `sort()` uses Timsort algorithm, which is a hybrid stable sorting algorithm derived from merge sort and insertion sort.

Time Complexity:  $O(n \log n)$  average case

Space Complexity:  $O(n)$

.....

# Create a copy to avoid modifying the original list

arr = arr.copy()

arr.sort()

return arr

def compare\_sorting\_algorithms(arr: List[int]) -> None:

.....

Compare bubble sort and Python's built-in sort() in terms of:

1. Correctness (both should produce the same result)

2. Performance (execution time)

.....

print("=" \* 70)

print("SORTING ALGORITHM COMPARISON")

print("=" \* 70)

print(f"\nArray size: {len(arr)} elements")

print(f"Original array (first 20 elements): {arr[:20]}")

if len(arr) > 20:

    print(f"Original array (last 10 elements): ...{arr[-10:]}")

# Test Bubble Sort (run multiple times for accuracy)

print("\n" + "-" \* 70)

print("BUBBLE SORT")

print("-" \* 70)

iterations = 10 if len(arr) < 1000 else 3

start\_time = time.time()

for \_ in range(iterations):

    bubble\_sorted = bubble\_sort(arr)

bubble\_time = (time.time() - start\_time) / iterations

```

print(f"Time taken: {bubble_time:.6f} seconds")
print(f"Sorted array (first 20 elements): {bubble_sorted[:20]}")
if len(bubble_sorted) > 20:
    print(f"Sorted array (last 10 elements): ...{bubble_sorted[-10:]}")

# Test Python's Built-in Sort (run multiple times for accuracy)

print("\n" + "-" * 70)
print("PYTHON'S BUILT-IN SORT()")
print("-" * 70)

iterations = 10 if len(arr) < 1000 else 3
start_time = time.time()
for _ in range(iterations):
    builtin_sorted = python_builtin_sort(arr)
    builtin_time = (time.time() - start_time) / iterations
    print(f"Time taken: {builtin_time:.6f} seconds")
    print(f"Sorted array (first 20 elements): {builtin_sorted[:20]}")
if len(builtin_sorted) > 20:
    print(f"Sorted array (last 10 elements): ...{builtin_sorted[-10:]}")

# Verify correctness

print("\n" + "-" * 70)
print("CORRECTNESS CHECK")
print("-" * 70)

is_correct = bubble_sorted == builtin_sorted
print(f"Both algorithms produce the same result: {is_correct}")

if is_correct:
    print("[OK] Both sorting algorithms are correct!")
else:
    print("[ERROR] Results don't match!")

# Performance comparison

print("\n" + "-" * 70)

```

```

print("PERFORMANCE COMPARISON")
print("-" * 70)
print(f"Bubble Sort time:  {bubble_time:.6f} seconds (average over {iterations} runs)")
print(f"Built-in Sort time: {builtin_time:.6f} seconds (average over {iterations} runs)")

if builtin_time > 0:
    speedup = bubble_time / builtin_time
    print(f"Speedup factor:   {speedup:.2f}x")
    if speedup > 1:
        print(f"\n-> Built-in sort() is {speedup:.2f}x faster than Bubble Sort")
    else:
        print(f"\n-> Bubble Sort is {1/speedup:.2f}x faster than built-in sort()")
else:
    print("\n-> Built-in sort() is extremely fast (time too small to measure accurately)")

print("\n" + "=" * 70)

def main():
    """Main function to demonstrate and compare sorting algorithms."""

    # Test with different array sizes
    test_sizes = [100, 500, 1000]
    for size in test_sizes:
        # Generate random array
        random_arr = [random.randint(1, 1000) for _ in range(size)]
        print(f"\n\n#{'=' * 70}")
        print(f"TEST CASE: Random array of {size} elements")
        print(f"{'=' * 70}")
        compare_sorting_algorithms(random_arr)

    # Test with already sorted array
    print(f"\n\n#{'=' * 70}")
    print("TEST CASE: Already sorted array (1000 elements)")
    print(f"{'=' * 70}")

```

```

sorted_arr = list(range(1, 1001))

compare_sorting_algorithms(sorted_arr)

# Test with reverse sorted array

print(f"\n\n{'#' * 70}")

print("TEST CASE: Reverse sorted array (1000 elements)")

print(f"{'#' * 70}")

reverse_arr = list(range(1000, 0, -1))

compare_sorting_algorithms(reverse_arr)

# Summary

print("\n\n" + "=" * 70)

print("SUMMARY")

print("=" * 70)

if __name__ == "__main__":
    main()

```

### **Output:**

Array size: 1000 elements

Original array (first 20 elements): [1000, 999, 998, 997, 996, 995, 994, 993, 992, 991, 990, 989, 988, 987, 986, 985, 984, 983, 982, 981]

Original array (last 10 elements): ...[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

### BUBBLE SORT

Time taken: 0.298021 seconds

Sorted array (first 20 elements): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Sorted array (last 10 elements): ...[991, 992, 993, 994, 995, 996, 997, 998, 999, 1000]

### PYTHON'S BUILT-IN SORT()

---

Time taken: 0.000000 seconds

Sorted array (first 20 elements): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Sorted array (last 10 elements): ...[991, 992, 993, 994, 995, 996, 997, 998, 999, 1000]

---

#### CORRECTNESS CHECK

---

Both algorithms produce the same result: True

[OK] Both sorting algorithms are correct!

---

#### PERFORMANCE COMPARISON

---

Bubble Sort time: 0.298021 seconds (average over 3 runs)

Built-in Sort time: 0.000000 seconds (average over 3 runs)

-> Built-in sort() is extremely fast (time too small to measure accurately)

The screenshot shows the AI Assistant Coding interface. On the left, there's a sidebar with 'Agents' and a list of tasks: 'Bubble sort and Python sort comparison' (Now - +167 Auto), 'CSV file statistics calcul...' (16m), 'Text file generation an...' (51m), and 'Python word frequency c...' (1h). A 'New Agent' button is also present. The main area has tabs for 'Agents', 'Editor', and 'AI ASSISTANT CODING'. The 'AI ASSISTANT CODING' tab is active, displaying a code editor with four files: 'word\_count(T1).py', 'Task-2.py', 'sort\_comparison.py', and 'sample.txt'. The 'sort\_comparison.py' file contains the following code:

```

1 import time
2 import random
3 from typing import List
4
5
6 def bubble_sort(arr: List[int]) -> List[int]:
7     """
8         Implement bubble sort algorithm.
9
10        Bubble sort repeatedly steps through the list, compares adjacent elements
11        and swaps them if they are in the wrong order. The pass through the list
12        is repeated until the list is sorted.
13
14        Time Complexity: O(n^2)
15        Space Complexity: O(1)
16    """
17    # Create a copy to avoid modifying the original list
18    arr = arr.copy()

```

Below the code editor, there's a 'Problems' tab, an 'Output' tab, a 'Debug Console' tab (which is selected), a 'Terminal' tab, and a 'Ports' tab. The 'Terminal' tab shows the following output:

```

-----
PERFORMANCE COMPARISON
-----
Bubble Sort time: 0.298021 seconds (average over 3 runs)
Built-in Sort time: 0.000000 seconds (average over 3 runs)

-> Built-in sort() is extremely fast (time too small to measure accurately)
-----
SUMMARY
-----
PS C:\Users\veema\OneDrive\Pictures\Desktop\HCL LAB\AI ASSISTANT CODING>

```

At the bottom, there are status indicators: 'Cursor Tab', 'Ln 163, Col 20', 'Spaces 4', 'UTF-8', and 'CPU F'.

## Explanation:

**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:  $O(n^2)$ ; 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  $O(n^2)$ ).