

AI-Ass-2.3

Name: T.Shivani

HT.NO:2303A51312

Batch:05

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.

Code:

"""

Read a text file and count word frequency.

Usage:

```
python word_frequency_counter.py path/to/file.txt
```

```
python word_frequency_counter.py path/to/file.txt --top 50
```

```
python word_frequency_counter.py path/to/file.txt --keywords error warn timeout
```

```
python word_frequency_counter.py path/to/file.txt --min-count 3 --top 100
```

"""

```
from __future__ import annotations
```

```
import argparse
```

```
import re
```

```
from collections import Counter
```

```
from pathlib import Path
```

```
from typing import Iterable
```

Words are sequences of letters/digits/underscore; this naturally ignores punctuation.

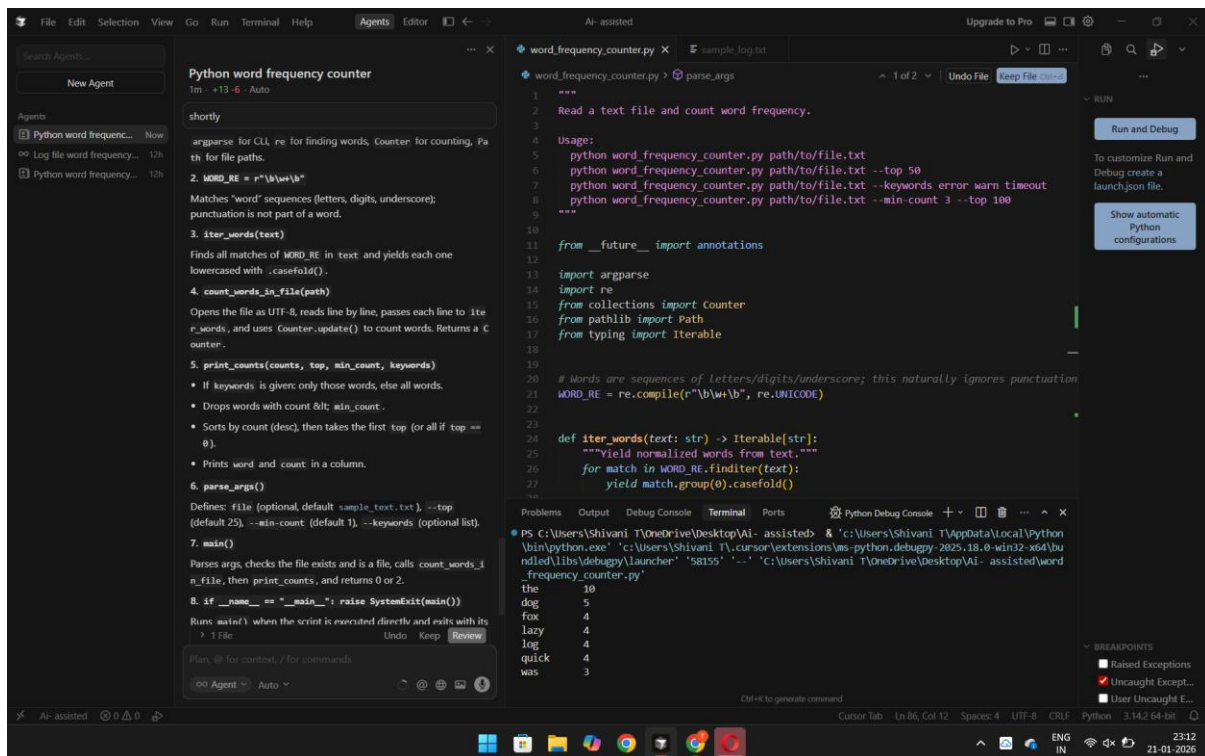
```
WORD_RE = re.compile(r"\b\w+\b", re.UNICODE)
```

```
def iter_words(text: str) -> Iterable[str]:
```

```
    """Yield normalized words from text."""
```

```
    for match in WORD_RE.finditer(text):
```

```
        yield match.group(0).casefold()
```



explanation: 1. Imports `argparse` for CLI, `re` for finding

words, `Counter` for counting, `Path` for file paths. 2. `WORD_RE = r"\b\w+\b"` Matches "word" sequences (letters, digits, underscore); punctuation is not part of a

word. 3. `iter_words(text)` Finds all matches of `WORD_RE` in text and yields each

one lowercased with `.casefold()`. 4. `count_words_in_file(path)` Opens the file as UTF-8, reads line by line, passes each line to `iter_words`, and uses `Counter.update()` to count words. Returns a Counter. 5. `print_counts(counts, top, min_count, keywords)`

- If keywords is given: only those words, else all words.
- Drops words with count `< min_count`.
- Sorts by count (desc), then takes the first top (or all if `top == 0`).
- Prints word and count in a column.

6. `parse_args()` Defines: `file` (optional, default `sample_text.txt`), `--top` (default 25), `--min-count` (default 1), `--keywords` (optional list).
7. `main()` Parses args, checks the file exists and is a file, calls `count_words_in_file`, then `print_counts`, and returns 0 or 2.
8. `if __name__ == "__main__":` raise `SystemExit(main())` Runs `main()` when the script is executed directly and exits with its return code.

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

Code:

```
def count_words_in_file(path: Path) -> Counter[str]:
    """Read a file and return a Counter of word frequencies."""
    counts: Counter[str] = Counter()

    # Use UTF-8 with replacement so "weird" bytes in logs won't crash the script.
    with path.open("r", encoding="utf-8", errors="replace") as f:
        for line in f:
            counts.update(iter_words(line))

    return counts


def print_counts(
    counts: Counter[str],
    *,
    top: int,
    min_count: int,
```

keywords: list[str] | None,

) -> None:

"""Print word counts, optionally filtered to specific keywords."""

if keywords:

Normalize keywords the same way we normalize text.

wanted = [k.casefold() for k in keywords]

items = [(k, counts.get(k, 0)) for k in wanted]

items = [(k, c) for k, c in items if c >= min_count]

items.sort(key=lambda kv: (-kv[1], kv[0]))

else:

items = [(w, c) for (w, c) in counts.items() if c >= min_count]

items.sort(key=lambda kv: (-kv[1], kv[0]))

if top > 0:

items = items[:top]

if not items:

print("No matches (check --min-count / --keywords).")

return

width = max(len(w) for (w, _) in items)

for word, count in items:

print(f"{word:<{width}} {count}")

def **parse_args()** -> argparse.Namespace:

p = argparse.ArgumentParser(*description*="Count word/keyword frequency in a text/log file.")

p.add_argument(

"file",

```

    type=Path,
    nargs="?",
    default=Path(__file__).parent / "sample_text.txt",
    help="Path to the input .txt/.log file (default: sample_text.txt in same folder)",
)

p.add_argument("--top", type=int, default=25, help="Show top N words (ignored with --
keywords). Use 0 for all.")

p.add_argument("--min-count", type=int, default=1, help="Only show words with count
>= this value")

p.add_argument(
    "--keywords",
    nargs="*",
    default=None,
    help="If provided, only show counts for these specific words (case-insensitive).",
)

return p.parse_args()

def main() -> int:
    args = parse_args()

    if not args.file.exists():
        print(f"File not found: {args.file}")
        return 2

    if not args.file.is_file():
        print(f"Not a file: {args.file}")
        return 2

    counts = count_words_in_file(args.file)

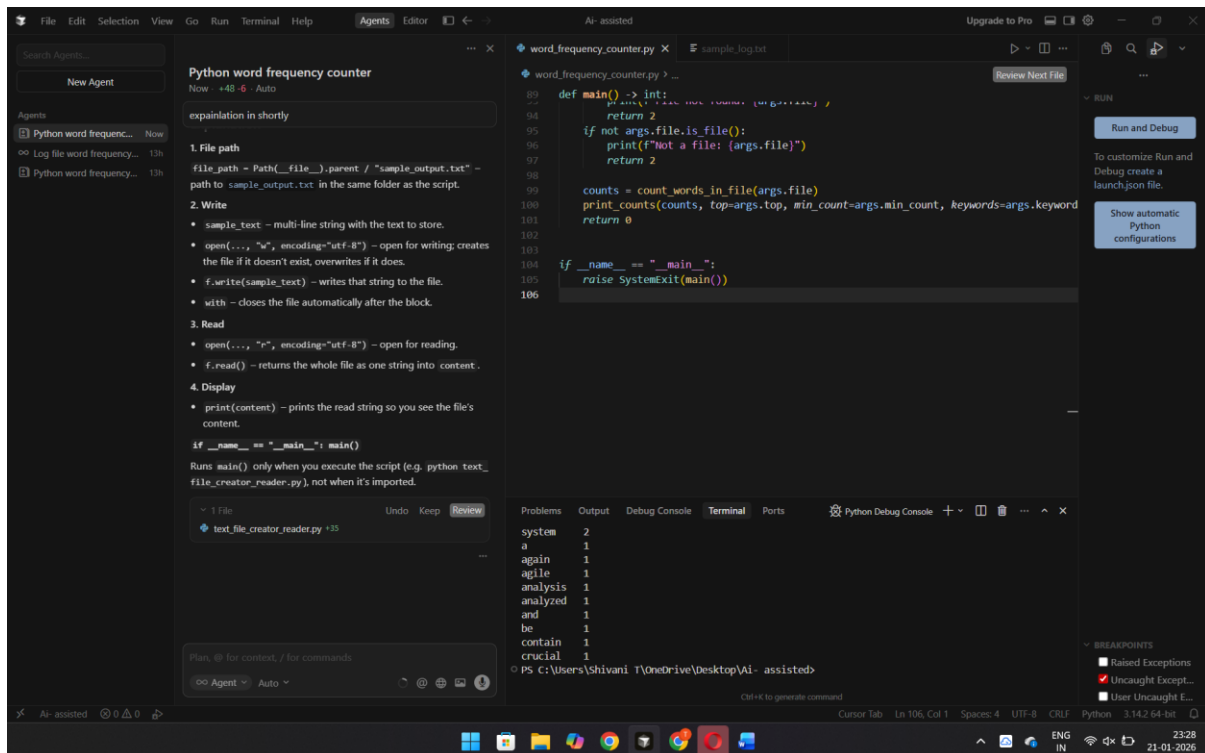
    print_counts(counts, top=args.top, min_count=args.min_count, keywords=args.keywords)

```

```
return 0
```

```
if __name__ == "__main__":
```

```
    raise SystemExit(main())
```



Explanation: 1. File path `file_path = Path(__file__).parent / "sample_output.txt"` – path to `sample_output.txt` in the same folder as the script. 2. Write

- `sample_text` – multi-line string with the text to store.
- `open(..., "w", encoding="utf-8")` – open for writing; creates the file if it doesn't exist, overwrites if it does.
- `f.write(sample_text)` – writes that string to the file.
- `with` – closes the file automatically after the block.

3. Read

- `open(..., "r", encoding="utf-8")` – open for reading.
- `f.read()` – returns the whole file as one string into `content`.

4. Display

- `print(content)` – prints the read string so you see the file's content.

if __name__ == "__main__": main()Runs main() only when you execute the script (e.g. python text_file_creator_reader.py), not when it's imported.

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.

Code: """

Read a CSV file and calculate mean, min, and max for numeric columns.

Usage:

```
python csv_stats.py [file.csv]
```

(default: sample_data.csv in same folder)

"""

```
import csv
```

```
import statistics
```

```
from pathlib import Path
```

```
def to_float(val: str) -> float | None:
```

```
    """Try to convert a string to float; return None if it fails."""
```

```
    val = val.strip()
```

```
    if not val:
```

```
        return None
```

```
    try:
```

```
        return float(val)
```

```
    except ValueError:
```

```
    return None
```

```
def get_numeric_columns(rows: list[dict]) -> list[str]:
```

```
    """Find columns where at least one value can be parsed as a number."""
```

```
    if not rows:
```

```
        return []
```

```
    numeric = []
```

```
    for col in rows[0]:
```

```
        values = [to_float(r.get(col, "")) for r in rows]
```

```
        if any(v is not None for v in values):
```

```
            numeric.append(col)
```

```
    return numeric
```

```
def compute_stats(rows: list[dict], col: str) -> dict:
```

```
    """Compute mean, min, max for a numeric column."""
```

```
    values = [to_float(r.get(col, "")) for r in rows]
```

```
    values = [v for v in values if v is not None]
```

```
    if not values:
```

```
        return {"mean": None, "min": None, "max": None}
```

```
    return {
```

```
        "mean": statistics.mean(values),
```

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

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

```
    }
```

```
def main(file_path: Path | None = None) -> int:
```

```
    if file_path is None:
```

```
        file_path = Path(__file__).parent / "sample_data.csv"
```

```
if not file_path.exists():  
    print(f"File not found: {file_path}")  
    print("Create sample_data.csv or pass a path: python csv_stats.py yourfile.csv")  
    return 2
```

```
with open(file_path, "r", encoding="utf-8", newline="") as f:  
    reader = csv.DictReader(f)  
    rows = list(reader)
```

```
if not rows:  
    print("CSV is empty or has no data rows.")  
    return 1
```

```
numeric_cols = get_numeric_columns(rows)  
if not numeric_cols:  
    print("No numeric columns found.")  
    return 1
```

```
print(f"File: {file_path}\n")  
print(f"{'Column':<16} {'Mean':>12} {'Min':>12} {'Max':>12}")  
print("-" * 56)
```

```
for col in numeric_cols:  
    s = compute_stats(rows, col)  
    mean = f"{s['mean']:.2f}" if s["mean"] is not None else "N/A"  
    min_ = f"{s['min']:.2f}" if s["min"] is not None else "N/A"  
    max_ = f"{s['max']:.2f}" if s["max"] is not None else "N/A"
```

```
print(f"col:<16} {mean:>12} {min_:>12} {max_:>12}")
```

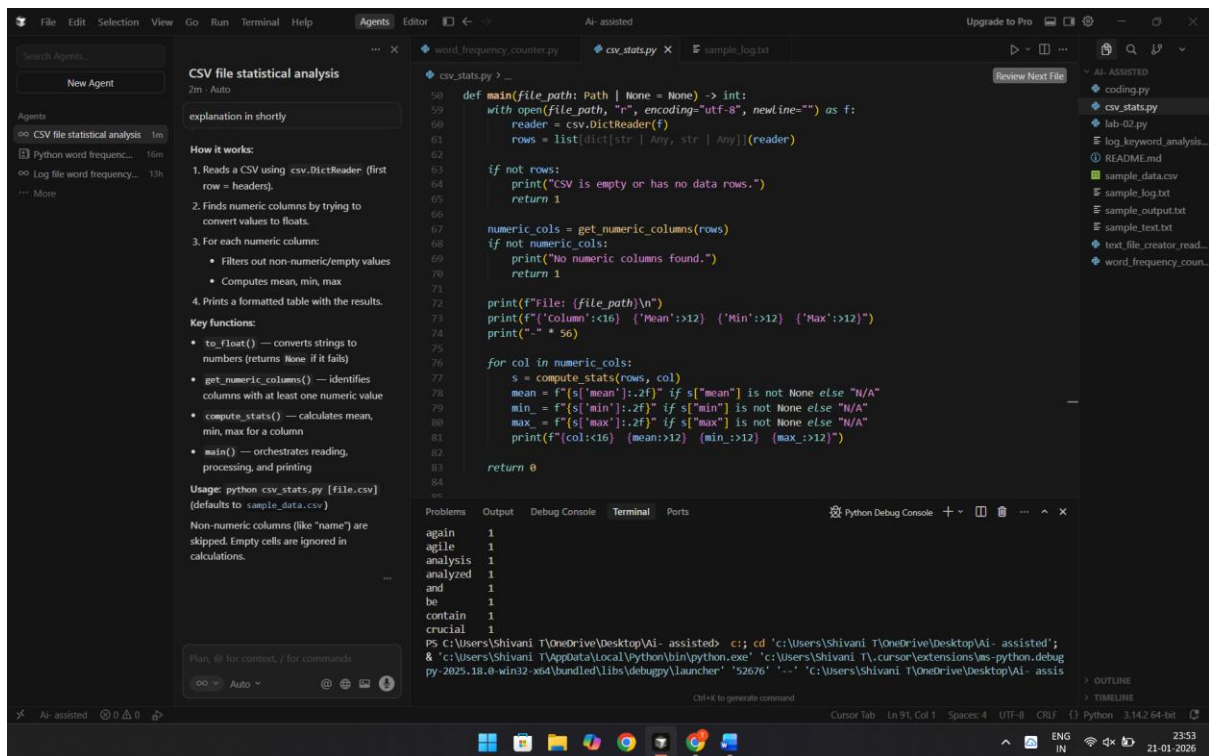
```
return 0
```

```
if __name__ == "__main__":
```

```
import sys
```

```
path = Path(sys.argv[1]) if len(sys.argv) > 1 else None
```

```
raise SystemExit(main(path))
```



Explanation:

How it works:

1. Reads a CSV using `csv.DictReader` (first row = headers).
2. Finds numeric columns by trying to convert values to floats.
3. For each numeric column:
 - Filters out non-numeric/empty values
 - Computes mean, min, max

4. Prints a formatted table with the results.

Key functions:

- `to_float()` — converts strings to numbers (returns `None` if it fails)
- `get_numeric_columns()` — identifies columns with at least one numeric value
- `compute_stats()` — calculates mean, min, max for a column
- `main()` — orchestrates reading, processing, and printing

Usage: `python csv_stats.py [file.csv]` (defaults to `sample_data.csv`) Non-numeric columns (like "name") are skipped. Empty cells are ignored in calculations.

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.

Code: """

Sorting Algorithm Comparison: Bubble Sort vs Python's Built-in `sort()`

This script demonstrates:

1. Bubble sort implementation
2. Python's built-in `sort()` method
3. Performance and correctness comparison

"""

import time

import random

```
from typing import List, Callable
```

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

```
    """
```

```
    Bubble Sort Algorithm Implementation
```

```
    Time Complexity:  $O(n^2)$ 
```

```
    Space Complexity:  $O(1)$ 
```

```
    Algorithm:
```

1. Compare adjacent elements
2. Swap if they are in wrong order
3. Repeat until no swaps are needed

```
    """
```

```
    # 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, list is sorted
```

```
        swapped = False
```

```
        # Last i elements are already in place
```

```
        for j in range(0, n - i - 1):
```

```
            # Compare adjacent elements
```

```
            if arr[j] > arr[j + 1]:
```

```
                # Swap if they are in wrong order
```

```
arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

```
swapped = True
```

```
# If no swaps were made, array is sorted
```

```
if not swapped:
```

```
    break
```

```
return arr
```

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

```
    """
```

```
    Python's Built-in sort() Method
```

```
    Uses Timsort algorithm (hybrid of merge sort and insertion sort)
```

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

```
    Space Complexity:  $O(n)$ 
```

```
    """
```

```
# Create a copy and sort in-place, then return
```

```
arr = arr.copy()
```

```
arr.sort()
```

```
return arr
```

```
def timed_sort(func: Callable, arr: List[int]) -> tuple[List[int], float]:
```

```
    """Measure execution time of a sorting function."""
```

```
    start_time = time.perf_counter()
```

```
    result = func(arr)
```

```
    end_time = time.perf_counter()
```

```
    elapsed_time = (end_time - start_time) * 1000 # Convert to milliseconds
```

```
return result, elapsed_time
```

```
def compare_sorts(test_data: List[int], label: str = "") -> None:
```

```
    """Compare bubble sort and Python's built-in sort on the same data."""
```

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

```
    if label:
```

```
        print(f"Test Case: {label}")
```

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

```
    print(f"Input size: {len(test_data)} elements")
```

```
    print(f"Input data (first 10): {test_data[:10]}...")
```

```
    # Test Bubble Sort
```

```
    bubble_result, bubble_time = timed_sort(bubble_sort, test_data)
```

```
    # Test Python's Built-in Sort
```

```
    builtin_result, builtin_time = timed_sort(python_builtin_sort, test_data)
```

```
    # Verify correctness
```

```
    is_correct = bubble_result == builtin_result
```

```
    is_sorted = bubble_result == sorted(test_data)
```

```
    # Display results
```

```
    print(f"\n{'Results':<25} {'Time (ms)':<15} {'Status':<20}")
```

```
    print("-" * 70)
```

```
    print(f"{'Bubble Sort':<25} {bubble_time:>12.4f} ms {'[OK] Correct' if is_correct else '[X] Incorrect'})")
```

```
    print(f"{'Python built-in sort()':<25} {builtin_time:>12.4f} ms {'[OK] Correct' if is_sorted else '[X] Incorrect'})")
```

if is_correct:

print(f"\n[OK] Both implementations produce identical results!")

print(f"[OK] Speed difference: {bubble_time / builtin_time:.2f}x slower (Bubble Sort)")

else:

print(f"\n[X] Results differ! Check implementation.")

print(f"\nSorted output (first 10): {bubble_result[:10]}...")

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

def main():

"""Run comparison tests with different data sizes and patterns."""

print("Sorting Algorithm Comparison")

print("=" * 70)

Test 1: Small random array

random.seed(42)

small_random = [random.randint(1, 100) for _ in range(50)]

compare_sorts(small_random, "Small Random Array (50 elements)")

Test 2: Medium random array

medium_random = [random.randint(1, 1000) for _ in range(500)]

compare_sorts(medium_random, "Medium Random Array (500 elements)")

Test 3: Large random array

large_random = [random.randint(1, 10000) for _ in range(5000)]

compare_sorts(large_random, "Large Random Array (5000 elements)")

Test 4: Already sorted array (best case for bubble sort optimization)

sorted_array = list(range(100))

compare_sorts(sorted_array, "Already Sorted Array (100 elements)")

Test 5: Reverse sorted array (worst case for bubble sort)

reverse_sorted = list(range(100, 0, -1))

compare_sorts(reverse_sorted, "Reverse Sorted Array (100 elements)")

Test 6: Array with duplicates

```
duplicates = [5, 2, 8, 2, 9, 1, 5, 5, 3, 2] * 10
```

```
compare_sorts(duplicates, "Array with Duplicates (100 elements)")
```

Summary

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

```
print("Summary")
```

```
print("="*70)
```

```
print("""
```

Key Differences:

1. TIME COMPLEXITY:

- Bubble Sort: $O(n^2)$ - quadratic time
- Python `sort()`: $O(n \log n)$ - linearithmic time

2. SPACE COMPLEXITY:

- Bubble Sort: $O(1)$ - sorts in-place (we copy for comparison)
- Python `sort()`: $O(n)$ - requires additional memory for Timsort

3. PERFORMANCE:

- Bubble Sort: Slower, especially for large datasets
- Python `sort()`: Much faster, highly optimized C implementation

4. USE CASES:

- Bubble Sort: Educational purposes, small datasets, when memory is limited
- Python `sort()`: Production code, large datasets, when performance matters

5. STABILITY:

- Both algorithms are stable (maintain relative order of equal elements)

6. ADAPTIVITY:

- Bubble Sort: Can be optimized to detect already-sorted arrays
- Python `sort()`: Highly adaptive, detects patterns in data

```
""")
```

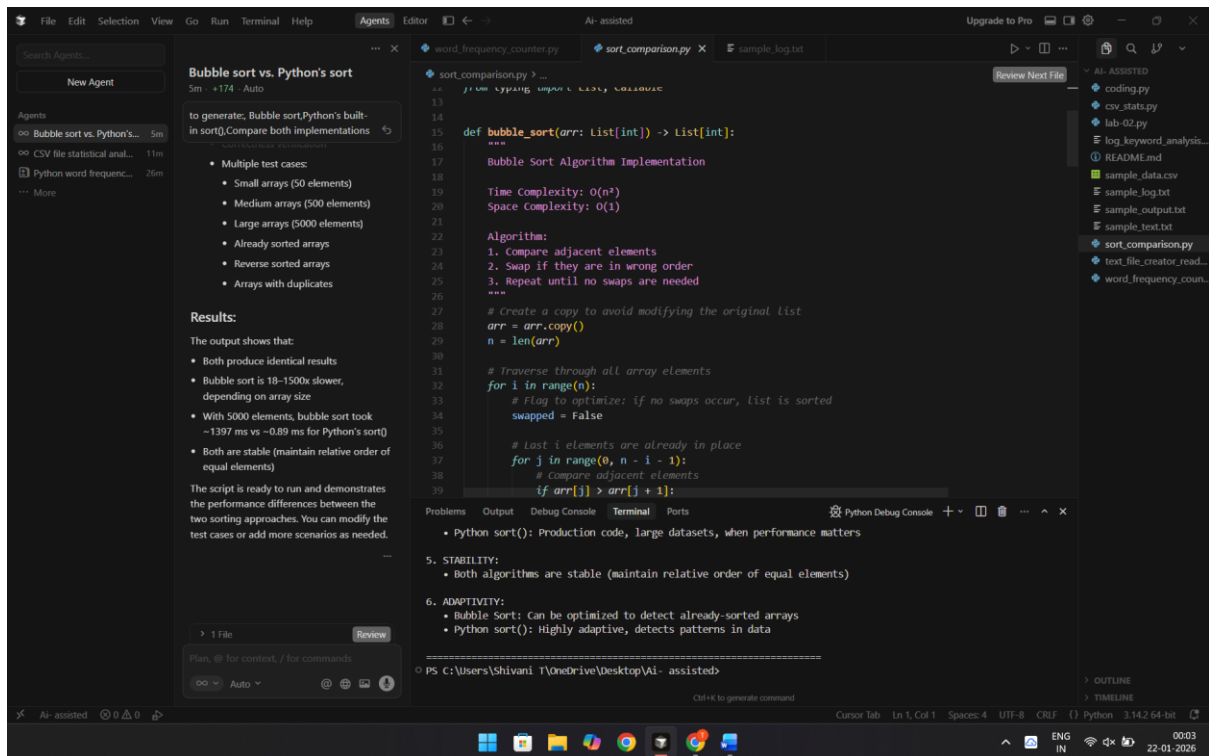
```

print("="*70)

if __name__ == "__main__":

    main()

```



Explanation: Bubble Sort:

- Compares adjacent elements and swaps if out of order
- Repeats until no swaps occur
- Time: $O(n^2)$ — slow for large arrays
- Space: $O(1)$ — sorts in place

Python's Built-in sort():

- Uses Timsort (merge + insertion sort hybrid)
- Time: $O(n \log n)$ — faster
- Space: $O(n)$ — needs extra memory
- Optimized C implementation

Key differences:

- Speed: `sort()` is much faster (often 100–1000x+ on large arrays)

- Complexity: Bubble sort is $O(n^2)$, `sort()` is $O(n \log n)$
- Use cases: Bubble sort is mainly educational; `sort()` is for production

Bottom line: Both produce correct results, but `sort()` is faster. Use bubble sort for learning; use `sort()` in real code. The script tests both on various datasets and shows the performance gap.