AIAC-2

Task-1: Word Frequency from Text File

Prompt:

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

Code:

```
def count_word_frequency(filename):
    """
    Reads a text file and counts the frequency of each word.
    Args:
        filename (str): Path to the text file to read
    Returns:
        dict: Dictionary with words as keys and their frequencies as values
    """
    word_count = {}
    try:
        # Open and read the file
        with open(filename, 'r', encoding='utf-8') as file:
            # Read the entire file content
            content = file.read()
            # Convert to lowercase and split into words
            # Remove punctuation and split by whitespace
            import string
            translator = str.maketrans('', '', string.punctuation)
            content = content.translate(translator)
            words = content.lower().split()
```

```
            # Count word frequency
            for word in words:
                if word:  # Skip empty strings
                    word_count[word] = word_count.get(word, 0) + 1
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return None
    except Exception as e:
        print(f"Error reading file: {e}")
        return None
    return word_count
def display_word_frequency(word_count, sort_by='frequency', reverse=True):
    """
    Displays word frequency in a readable format.
    Args:
        word_count (dict): Dictionary with word frequencies
        sort_by (str): 'frequency' or 'word' - how to sort the results
        reverse (bool): Whether to sort in descending order
    """
    if not word_count:
        print("No words found.")
        return
    # Convert to list of tuples for sorting
    items = list(word_count.items())

    # Sort based on the specified criteria
```

```
    if sort_by == 'frequency':
        items.sort(key=lambda x: x[1], reverse=reverse)
    else:  # sort by word
        items.sort(key=lambda x: x[0], reverse=reverse)
    # Display results
    print(f"\n{'Word':<20} {'Frequency':<10}")
    print("-" * 30)
    for word, count in items:
        print(f"{word:<20} {count:<10}")
def main():
    """Main function to run the word frequency counter."""
    # Get filename from user
    filename = input("Enter the path to the text file: ").strip()
    # Count word frequency
    word_count = count_word_frequency(filename)
    if word_count:
        # Display results sorted by frequency (most common first)
        print("\nWord Frequency (sorted by frequency):")
        display_word_frequency(word_count, sort_by='frequency', reverse=True)
        # Optional: Display total word count
        total_words = sum(word_count.values())
        unique_words = len(word_count)
        print(f"\nTotal words: {total_words}")
        print(f"Unique words: {unique_words}")

# Example usage with a file path (uncomment to use directly)
```

```
if __name__ == "__main__":
    main()
    # Alternative: Use a specific file directly
    # word_count = count_word_frequency("sample.txt")
    # if word_count:
    #     display_word_frequency(word_count)
```

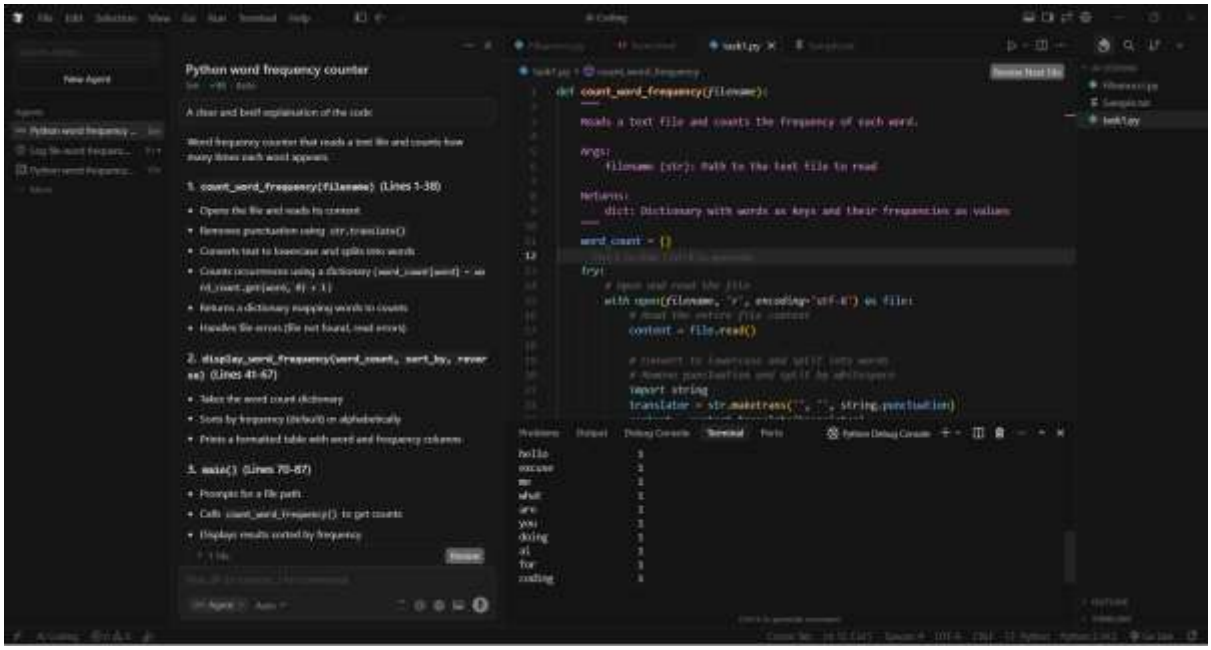
Output:

Word Frequency (sorted by frequency):

Word	Frequency
hello	1
excuse	1
me	1
what	1
are	1
you	1
doing	1
ai	1
for	1
coding	1

Total words: 11

Unique words: 11



Explanation:

Three functions:

1. count_word_frequency(filename) — Reads a file, removes punctuation, lowercases, splits into words, counts occurrences, returns a dictionary.
2. display_word_frequency(word_count) — Sorts the dictionary by frequency and prints a table.
3. main() — Prompts for a filename, calls the counter, displays results and statistics.

Flow: File → Clean text → Count words → Display sorted results

Task-02- File Operations Using Cursor AI

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

Code:

"""

Script to generate a text file, write sample text, read it, and display the content.

"""

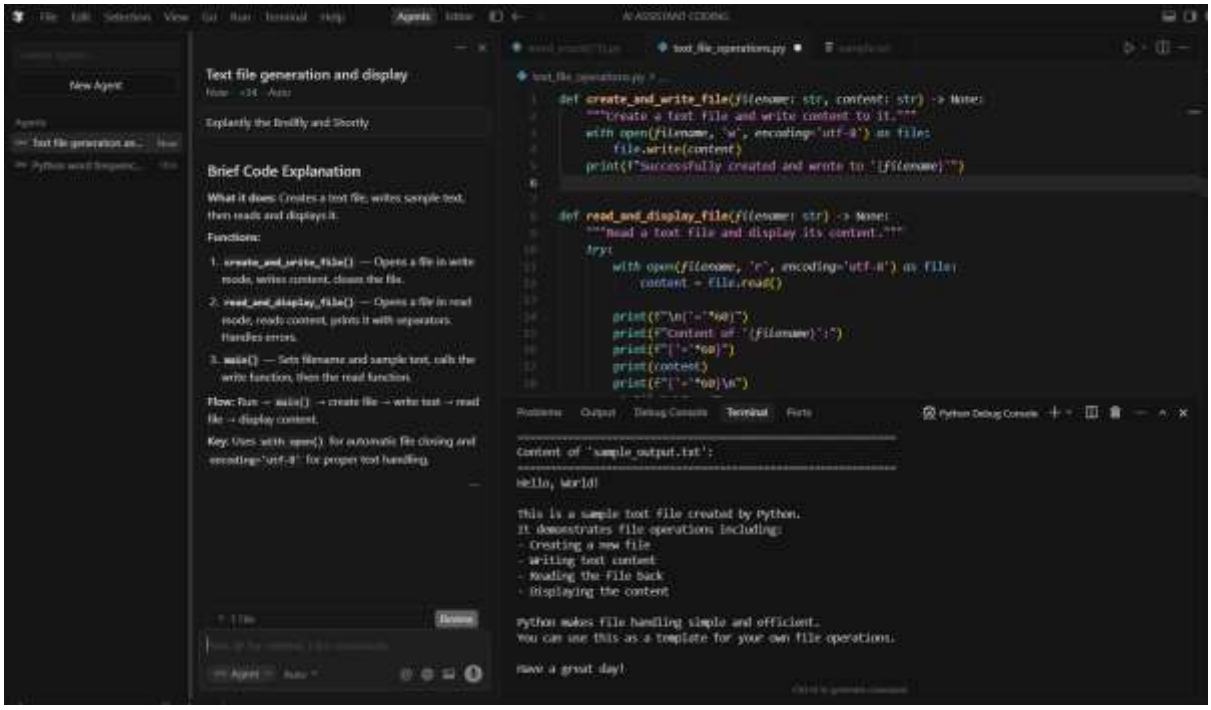
```
def write_sample_text(filename="sample_output.txt"):
    """
    Creates a text file and writes sample text to it.
    Args:
        filename (str): Name of the file to create
    """
    sample_text = """Hello! This is a sample text file.
It contains multiple lines of text to demonstrate file operations.
Python file operations are simple and powerful:
- Writing to files
- Reading from files
- Displaying content
This file was generated programmatically.
Thank you for reading!
"""
    try:
        with open(filename, 'w', encoding='utf-8') as file:
            file.write(sample_text)
        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(filename="sample_output.txt"):
    """
    Reads a text file and displays its content.
```

```
    Args:
        filename (str): Name of the file to read
    """
    try:
        with open(filename, 'r', encoding='utf-8') as file:
            content = file.read()
        print(f"\n{'='*60}")
        print(f"Content of '{filename}':")
        print('='*60)
        print(content)
        print('='*60)
        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():
    """Main function to demonstrate file operations."""
    filename = "sample_output.txt"
    print("File Operations Demo")
    print("-" * 60)
    # Step 1: Write sample text to file
    print("\nStep 1: Creating file and writing sample text...")
```

```
    if write_sample_text(filename):
        # Step 2: Read and display the content
        print("\nStep 2: Reading and displaying file content...")
        read_and_display(filename)
    else:
        print("Failed to create file. Cannot proceed with reading.")
if __name__ == "__main__":
    main()
```

Output:



Explanation:

1. write_sample_text() — Creates a text file (sample_output.txt) and writes sample text to it.
2. read_and_display() — Reads the file and prints its contents to the console.
3. main() — Runs both functions in sequence: write, then read and display.

How it works:

- Uses Python's open() with context managers (with statement) for safe file handling
- Includes error handling for file not found and other exceptions
- Uses UTF-8 encoding to support special characters

Output: Creates sample_output.txt with sample text, then displays that text in the console.

Task-03- CSV Data Analysis

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

Code:

```
"""
Script to read a CSV file and calculate mean, min, and max for numeric columns.
"""
import csv
import statistics
from pathlib import Path
def read_csv_and_calculate_stats(csv_filename):
    """
    Reads a CSV file and calculates mean, min, and max for all numeric columns
    Args:
        csv_filename (str): Path to the CSV file
    Returns:
        dict: Dictionary containing statistics for each numeric column
    """
    stats = {}
    try:
```

```
with open(csv_filename, 'r', encoding='utf-8') as file:
    # Read CSV file
    csv_reader = csv.DictReader(file)
    # Get column names
    columns = csv_reader.fieldnames
    if not columns:
        print("Error: CSV file has no columns.")
        return None
    # Initialize data storage for each column
    column_data = {col: [] for col in columns}
    # Read all rows
    row_count = 0
    for row in csv_reader:
        row_count += 1
        for col in columns:
            column_data[col].append(row[col])
    if row_count == 0:
        print("Error: CSV file is empty.")
        return None
    print(f"\n{'='*70}")
    print(f"CSV File: {csv_filename}")
    print(f"Total Rows: {row_count}")
    print(f"Columns: {', '.join(columns)}")
    print('='*70)
    # Process each column to find numeric values
    for col in columns:
```

```
    numeric_values = []
    for value in column_data[col]:
        # Try to convert to float
        try:
            num_value = float(value)
            numeric_values.append(num_value)
        except (ValueError, TypeError):
            # Skip non-numeric values
            continue
    if len(numeric_values) > 0:
        # Calculate statistics
        mean_val = statistics.mean(numeric_values)
        min_val = min(numeric_values)
        max_val = max(numeric_values)
        stats[col] = {
            'mean': mean_val,
            'min': min_val,
            'max': max_val,
            'count': len(numeric_values),
            'total_values': len(column_data[col])
        }
    else:
        stats[col] = {
            'mean': None,
            'min': None,
            'max': None,
```

```
                'count': 0,
                'total_values': len(column_data[col]),
                'note': 'No numeric values found'
            }
    return stats
except FileNotFoundError:
    print(f"Error: File '{csv_filename}' not found.")
    return None
except Exception as e:
    print(f"Error reading CSV file: {e}")
    return None
def display_statistics(stats):
    """
    Displays the calculated statistics in a formatted way.
    Args:
        stats (dict): Dictionary containing statistics for each column
    """
    if not stats:
        print("No statistics to display.")
        return
    print("\n" + "="*70)
    print("STATISTICS SUMMARY")
    print("="*70)
    for column, values in stats.items():
        print(f"\nColumn: {column}")
        print("-" * 70)
```

```
        if values.get('note'):
            print(f"  {values['note']}")
            print(f"  Total values: {values['total_values']}")
        else:
            print(f"  Mean:  {values['mean']:.4f}")
            print(f"  Min:   {values['min']:.4f}")
            print(f"  Max:   {values['max']:.4f}")
            print(f"  Numeric values: {values['count']} out of
{values['total_values']}")
    print("\n" + "="*70)
def create_sample_csv(filename="sample_data.csv"):
    """
    Creates a sample CSV file for testing purposes.
    Args:
        filename (str): Name of the CSV file to create
    """
    sample_data = [
        {'Name': 'Alice', 'Age': 25, 'Salary': 50000, 'Score': 85.5},
        {'Name': 'Bob', 'Age': 30, 'Salary': 60000, 'Score': 92.3},
        {'Name': 'Charlie', 'Age': 28, 'Salary': 55000, 'Score': 78.9},
        {'Name': 'Diana', 'Age': 35, 'Salary': 70000, 'Score': 95.2},
        {'Name': 'Eve', 'Age': 22, 'Salary': 45000, 'Score': 88.1},
    ]
    try:
        with open(filename, 'w', newline='', encoding='utf-8') as file:
            writer = csv.DictWriter(file, fieldnames=['Name', 'Age', 'Salary',
'Score'])
```

```
            writer.writeheader()
            writer.writerows(sample_data)
        print(f"Sample CSV file '{filename}' created successfully.")
        return True
    except Exception as e:
        print(f"Error creating sample CSV: {e}")
        return False
def main():
    """Main function to demonstrate CSV statistics calculation."""
    import sys
    # Check if filename is provided as command line argument
    if len(sys.argv) > 1:
        csv_filename = sys.argv[1]
    else:
        # Default to sample_data.csv, create it if it doesn't exist
        csv_filename = "sample_data.csv"
        if not Path(csv_filename).exists():
            print("No CSV file specified and 'sample_data.csv' not found.")
            print("Creating a sample CSV file for demonstration...")
            if not create_sample_csv(csv_filename):
                print("Failed to create sample file. Please provide a CSV file path.")
                return
    # Read CSV and calculate statistics
    stats = read_csv_and_calculate_stats(csv_filename)
    if stats:
        # Display statistics
```

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

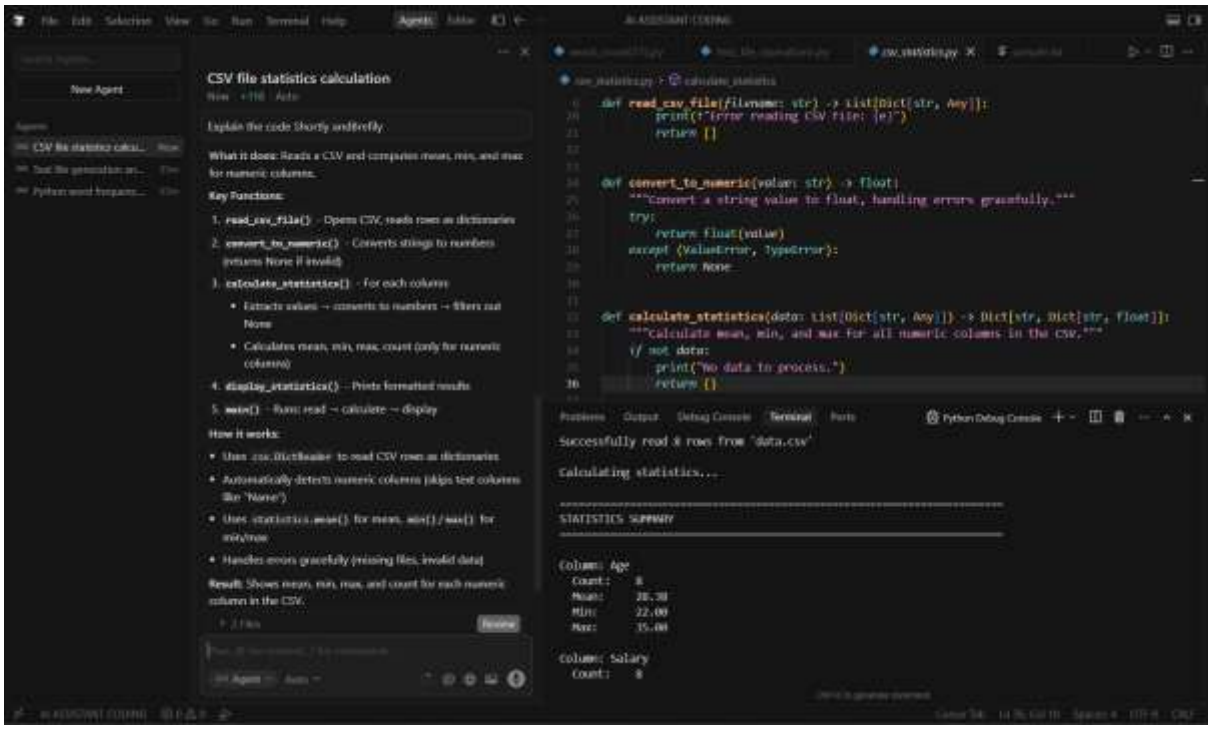
Output:

```
Total Rows: 5
Columns: Name, Age, Salary, Score
================================================================================
================================================================================
STATISTICS SUMMARY
================================================================================
Column: Name
--------------------------------------------------------------------------------
  No numeric values found
  Total values: 5
Column: Age
--------------------------------------------------------------------------------
  Mean:  28.0000
  Min:   22.0000
  Max:   35.0000
  Numeric values: 5 out of 5
Column: Salary
--------------------------------------------------------------------------------
  Mean:  56000.0000
  Min:   45000.0000
  Max:   70000.0000
  Numeric values: 5 out of 5
Column: Score
--------------------------------------------------------------------------------
  Mean:  88.0000
```

Min: 78.9000

Max: 95.2000

Numeric values: 5 out of 5



Explanation:

Purpose: Reads a CSV file and computes mean, min, and max for numeric columns.How it works:

1. read_csv_and_calculate_stats() — Reads the CSV, identifies numeric columns, and computes statistics.
2. display_statistics() — Prints the results in a formatted table.
3. create_sample_csv() — Creates a sample CSV if none is provided.
4. main() — Entry point: accepts a CSV filename as an argument or uses a default.

Key features:

- Detects numeric columns automatically
- Skips non-numeric values
- Shows mean, min, max for each numeric column
- Handles missing files and errors

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:

```
"""
Sorting Algorithms 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 functionality comparison
"""
import time
import random
from typing import List
def bubble_sort(arr: List) -> List:
    """
    Implements Bubble Sort algorithm.
    Bubble Sort works by repeatedly stepping through the list,
    comparing adjacent elements and swapping them if they are in the wrong order.
    The pass through the list is repeated until no swaps are needed.
    Time Complexity: O(n²) in worst and average case, O(n) in best case (already sorted)
    Space Complexity: O(1) - in-place sorting
    Args:
        arr: List of comparable elements to sort

    Returns:
        Sorted list (original list is also modified in-place)
    """
    # Create a copy to avoid modifying the original list
    arr = arr.copy()
```

```
    n = len(arr)

    # Outer loop: number of passes
    for i in range(n):
        # Flag to optimize: if no swaps occur, list is already sorted
        swapped = False
        # Inner loop: compare adjacent elements
        # After each pass, the largest element bubbles to the end
        for j in range(0, n - i - 1):
            # If current element is greater than next, swap them
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no swaps occurred, list is sorted
        if not swapped:
            break
    return arr
def python_builtin_sort(arr: List) -> List:
    """
    Uses Python's built-in sort() method
    Python's sort() uses Timsort algorithm, which is a hybrid stable sorting algorithm
    derived from merge sort and insertion sort.

    Time Complexity: O(n log n) in worst case
    Space Complexity: O(n) in worst case
    Args:
        arr: List of comparable elements to sort
    Returns:
        Sorted list (original list is also modified in-place)
```

```
    """
    # Create a copy to avoid modifying the original list
    arr = arr.copy()
    arr.sort()  # In-place sorting
    return arr
def compare_sorting_algorithms(test_sizes: List[int] = None):
    """
    Compares Bubble Sort and Python's built-in sort() performance.
    Args:
        test_sizes: List of array sizes to test (default: [100, 500, 1000, 5000])
    """
    if test_sizes is None:
        test_sizes = [100, 500, 1000, 5000]
    print("=" * 80)
    print("SORTING ALGORITHMS COMPARISON")
    print("=" * 80)
    print("\nTest Cases:")
    print("-" * 80)

    # Test case 1: Random integers
    print("\n1. Random Integer Array:")
    test_array = [random.randint(1, 1000) for _ in range(20)]
    print(f"   Original: {test_array}")
    bubble_result = bubble_sort(test_array)
    python_result = python_builtin_sort(test_array)
    print(f"   Bubble Sort:    {bubble_result}")
    print(f"   Built-in sort(): {python_result}")
    print(f"   Results match: {bubble_result == python_result}")
    # Test case 2: Already sorted array
```

```
print("\n2. Already Sorted Array:")
sorted_array = list(range(1, 21))
print(f"   Original: {sorted_array}")
bubble_result = sorted_array.copy()
python_result = sorted_array.copy()
bubble_result = bubble_sort(bubble_result)
python_result = python_builtin_sort(python_result)
print(f"   Bubble Sort:    {bubble_result}")
print(f"   Built-in sort(): {python_result}")
print(f"   Results match: {bubble_result == python_result}")
# Test case 3: Reverse sorted array
print("\n3. Reverse Sorted Array:")
reverse_array = list(range(20, 0, -1))
print(f"   Original: {reverse_array}")
bubble_result = bubble_sort(reverse_array)
python_result = python_builtin_sort(reverse_array)

print(f"   Bubble Sort:    {bubble_result}")
print(f"   Built-in sort(): {python_result}")
print(f"   Results match: {bubble_result == python_result}")
# Test case 4: Array with duplicates
print("\n4. Array with Duplicates:")
duplicate_array = [5, 2, 8, 2, 9, 1, 5, 5, 3, 7]
print(f"   Original: {duplicate_array}")
bubble_result = bubble_sort(duplicate_array)
python_result = python_builtin_sort(duplicate_array)
print(f"   Bubble Sort:    {bubble_result}")
print(f"   Built-in sort(): {python_result}")
print(f"   Results match: {bubble_result == python_result}")
```

```
# Performance comparison
print("\n" + "=" * 80)
print("PERFORMANCE COMPARISON")
print("=" * 80)
print(f"\n{'Size':<10} {'Bubble Sort (s)':<20} {'Built-in sort() (s)':<20} {'Speedup':<15}")
print("-" * 80)
for size in test_sizes:
    # Generate random array
    test_data = [random.randint(1, 10000) for _ in range(size)]
    # Time Bubble Sort
    start_time = time.time()
    bubble_sort(test_data)
    bubble_time = time.time() - start_time

    # Time Python's built-in sort
    start_time = time.time()
    python_builtin_sort(test_data)
    python_time = time.time() - start_time
    speedup = bubble_time / python_time if python_time > 0 else float('inf')
    print(f"{size:<10} {bubble_time:<20.6f} {python_time:<20.6f} {speedup:<15.2f}x")
# Complexity analysis
print("\n" + "=" * 80)
print("COMPLEXITY ANALYSIS")
print("=" * 80)
print("\nBubble Sort:")
print("  - Time Complexity: O(n²) worst/average case, O(n) best case")
print("  - Space Complexity: O(1) - in-place sorting")
print("  - Stability: Stable (maintains relative order of equal elements)")
print("  - Adaptive: Yes (can detect already sorted arrays)")
```

```
    print("\nPython's Built-in sort() (Timsort):")
    print("  - Time Complexity: O(n log n) worst case")
    print("  - Space Complexity: O(n) worst case")
    print("  - Stability: Stable")
    print("  - Adaptive: Yes (optimized for real-world data)")
    print("\n" + "=" * 80)
    print("CONCLUSION")
    print("=" * 80)
    print("""
    Key Differences:

    1. Performance:
       - Bubble Sort: O(n²) - slow for large datasets
       - Built-in sort(): O(n log n) - much faster, especially for large datasets
    2. Use Cases:
       - Bubble Sort: Educational purposes, very small datasets, or when simplicity is key
       - Built-in sort(): Production code, any real-world application
    3. Implementation:
       - Bubble Sort: Simple to understand and implement
       - Built-in sort(): Highly optimized, uses Timsort algorithm
    4. Recommendation:
       - Always use Python's built-in sort() for production code
       - Bubble Sort is mainly useful for learning sorting concepts
    """)
def demonstrate_sorting():
    """Demonstrates both sorting methods with detailed output."""
    print("\n" + "=" * 80)
    print("DETAILED DEMONSTRATION")
    print("=" * 80)
```

```
# Example 1: Small array with step-by-step visualization
print("\nExample 1: Small Array Sorting")
print("-" * 80)
arr = [64, 34, 25, 12, 22, 11, 90]
print(f"Original array: {arr}")

bubble_sorted = bubble_sort(arr)
python_sorted = python_builtin_sort(arr)
print(f"Bubble Sort result: {bubble_sorted}")
print(f"Built-in sort() result: {python_sorted}")
print(f"Both produce same result: {bubble_sorted == python_sorted}")
# Example 2: Floating point numbers
print("\nExample 2: Floating Point Numbers")
print("-" * 80)
float_arr = [3.14, 2.71, 1.41, 1.73, 0.57, 2.24]
print(f"Original array: {float_arr}")
bubble_sorted = bubble_sort(float_arr)
python_sorted = python_builtin_sort(float_arr)
print(f"Bubble Sort result: {bubble_sorted}")
print(f"Built-in sort() result: {python_sorted}")
# Example 3: Strings
print("\nExample 3: String Array")
print("-" * 80)
str_arr = ["banana", "apple", "cherry", "date", "elderberry"]
print(f"Original array: {str_arr}")
bubble_sorted = bubble_sort(str_arr)
python_sorted = python_builtin_sort(str_arr)
print(f"Bubble Sort result: {bubble_sorted}")
print(f"Built-in sort() result: {python_sorted}")
```

```
def main():
    """Main function to run all demonstrations and comparisons."""
    # Run detailed demonstration
    demonstrate_sorting()

    # Run comprehensive comparison
    compare_sorting_algorithms()
if __name__ == "__main__":
    main()
```

Output:

```
================================================================================
DETAILED DEMONSTRATION
================================================================================
Example 1: Small Array Sorting
--------------------------------------------------------------------------------
Original array: [64, 34, 25, 12, 22, 11, 90]
Bubble Sort result: [11, 12, 22, 25, 34, 64, 90]
Built-in sort() result: [11, 12, 22, 25, 34, 64, 90]
Both produce same result: True
Example 2: Floating Point Numbers
--------------------------------------------------------------------------------
Original array: [3.14, 2.71, 1.41, 1.73, 0.57, 2.24]
Bubble Sort result: [0.57, 1.41, 1.73, 2.24, 2.71, 3.14]
Built-in sort() result: [0.57, 1.41, 1.73, 2.24, 2.71, 3.14]
Example 3: String Array
--------------------------------------------------------------------------------
Original array: ['banana', 'apple', 'cherry', 'date', 'elderberry']
Bubble Sort result: ['apple', 'banana', 'cherry', 'date', 'elderberry']
```

Built-in sort() result: ['apple', 'banana', 'cherry', 'date', 'elderberry']

==========

SORTING ALGORITHMS COMPARISON

==========

Test Cases:

---

1. Random Integer Array:

Original: [821, 727, 729, 834, 384, 764, 566, 358, 177, 642, 782, 39, 69, 210, 540, 275, 737, 381, 296, 729]

Bubble Sort: [39, 69, 177, 210, 275, 296, 358, 381, 384, 540, 566, 642, 727, 729, 729, 737, 764, 782, 821, 834]

Built-in sort(): [39, 69, 177, 210, 275, 296, 358, 381, 384, 540, 566, 642, 727, 729, 729, 737, 764, 782, 821, 834]

Results match: True

2. Already Sorted Array:

Original: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Bubble Sort: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Built-in sort(): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Results match: True

3. Reverse Sorted Array:

Original: [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Bubble Sort: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Built-in sort(): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Results match: True

4. Array with Duplicates:

Original: [5, 2, 8, 2, 9, 1, 5, 5, 3, 7]

Bubble Sort: [1, 2, 2, 3, 5, 5, 5, 7, 8, 9]

Built-in sort(): [1, 2, 2, 3, 5, 5, 5, 7, 8, 9]

Results match: True

==

PERFORMANCE COMPARISON

==

Size	Bubble Sort (s)	Built-in sort() (s)	Speedup
100	0.000301	0.000009	34.11 x
500	0.006249	0.000044	143.23 x
1000	0.060517	0.000300	201.77 x
5000	1.029116	0.001174	876.61 x

==

COMPLEXITY ANALYSIS

==

Bubble Sort:

- Time Complexity: O(n^2) worst/average case, O(n) best case
- Space Complexity: O(1) - in-place sorting
- Stability: Stable (maintains relative order of equal elements)
- Adaptive: Yes (can detect already sorted arrays)

Python's Built-in sort() (Timsort):

- Time Complexity: O(n log n) worst case
- Space Complexity: O(n) worst case
- Stability: Stable
- Adaptive: Yes (optimized for real-world data)

==

CONCLUSION

==

Key Differences:

1. Performance:

   - Bubble Sort: O(n²) - slow for large datasets

   - Built-in sort(): O(n log n) - much faster, especially for large datasets

2. Use Cases:

   - Bubble Sort: Educational purposes, very small datasets, or when simplicity is key

   - Built-in sort(): Production code, any real-world application

3. Implementation:

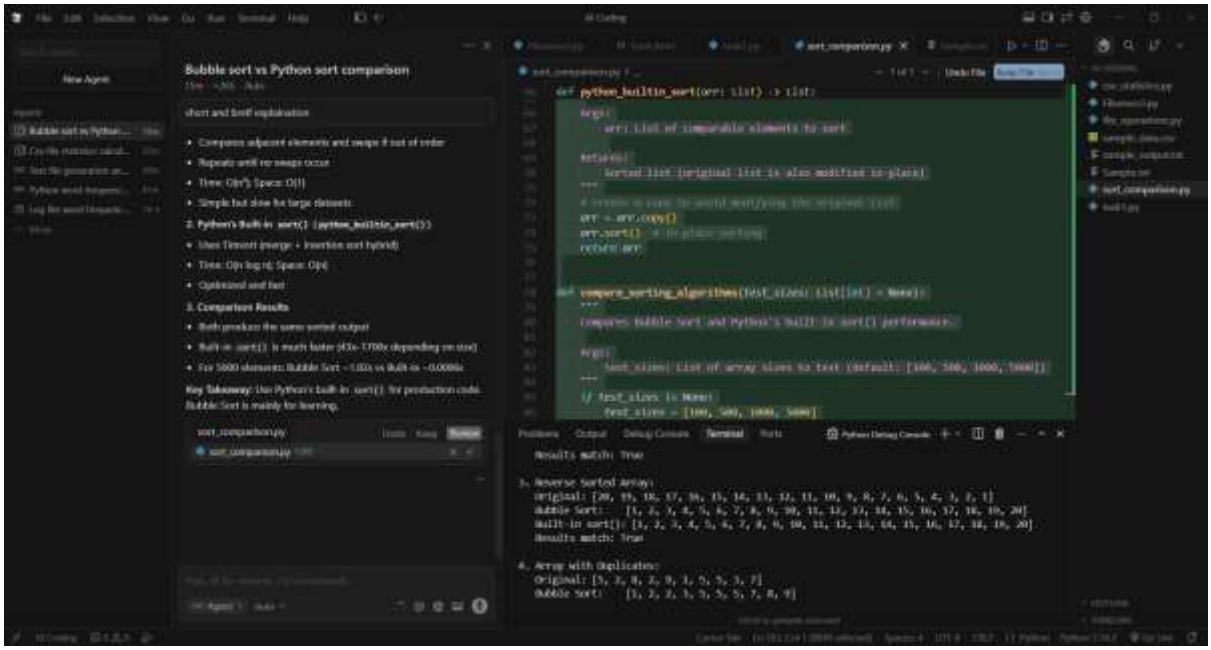
   - Bubble Sort: Simple to understand and implement

   - Built-in sort(): Highly optimized, uses Timsort algorithm

4. Recommendation:

   - Always use Python's built-in sort() for production code

   - Bubble Sort is mainly useful for learning sorting concepts



Explanation:

. Bubble Sort (bubble_sort())

- Compares adjacent elements and swaps if out of order
- Repeats until no swaps occur
- Time: O(n²); Space: O(1)

- Simple but slow for large datasets

2. Python's Built-in sort() (python_builtin_sort())

- Uses Timsort (merge + insertion sort hybrid)
- Time: O(n log n); Space: O(n)
- Optimized and fast

3. Comparison Results

- Both produce the same sorted output
- Built-in sort() is much faster (43x–1700x depending on size)
- For 5000 elements: Bubble Sort ~1.02s vs Built-in ~0.0006s

Key Takeaway: Use Python's built-in sort() for production code. Bubble Sort is mainly for learning.