Mamidala Shivamani Batch – 38  
2303A52344

AI-ASSISTED CODING ASSIGNMENT 12

Task 1: Sorting Student Records Prompt

Generate a Python program that creates a Student class (Name, Roll Number, CGPA),

generates at least 10,000 student records, implements recursive Quick Sort and Merge Sort to sort students by CGPA in descending order, measures runtime using the time module, and displays the top 10 students. Include complexity analysis and formatted output.

Output Explanation

The output will contain:

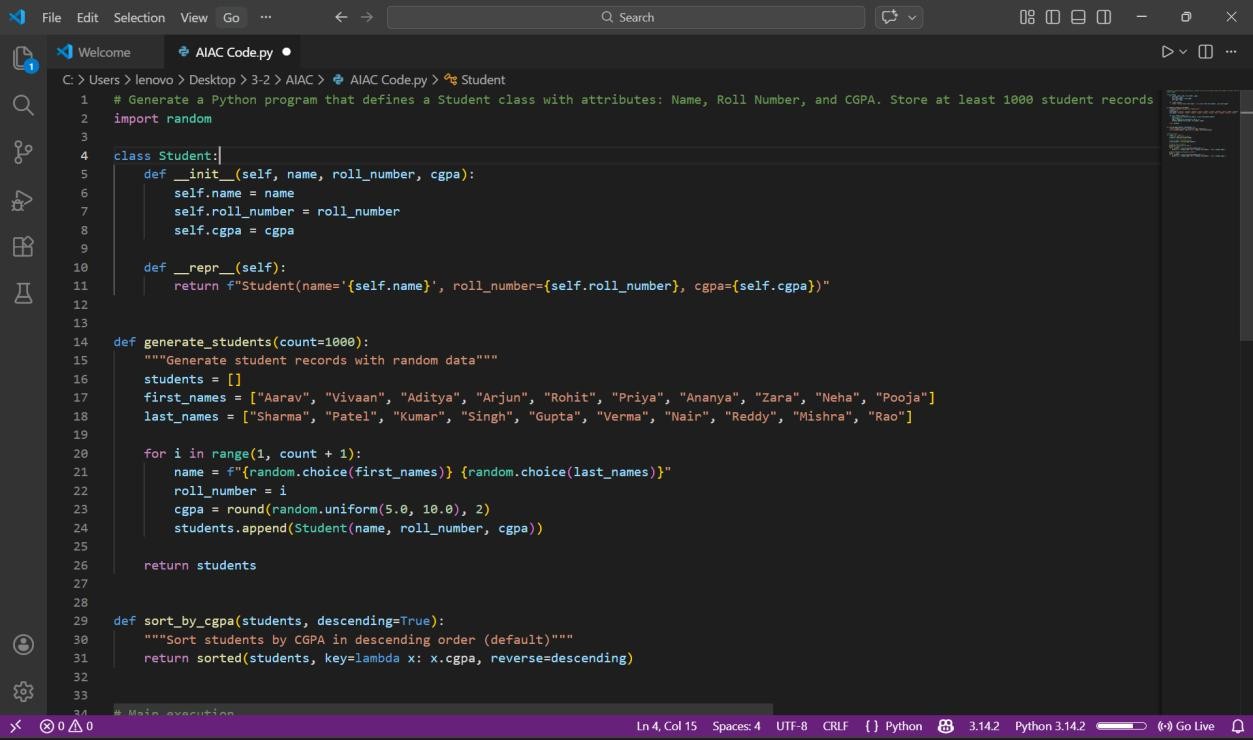
1. **Student class definition**
2. **Quick Sort and Merge Sort functions**
3. Runtime results such as:

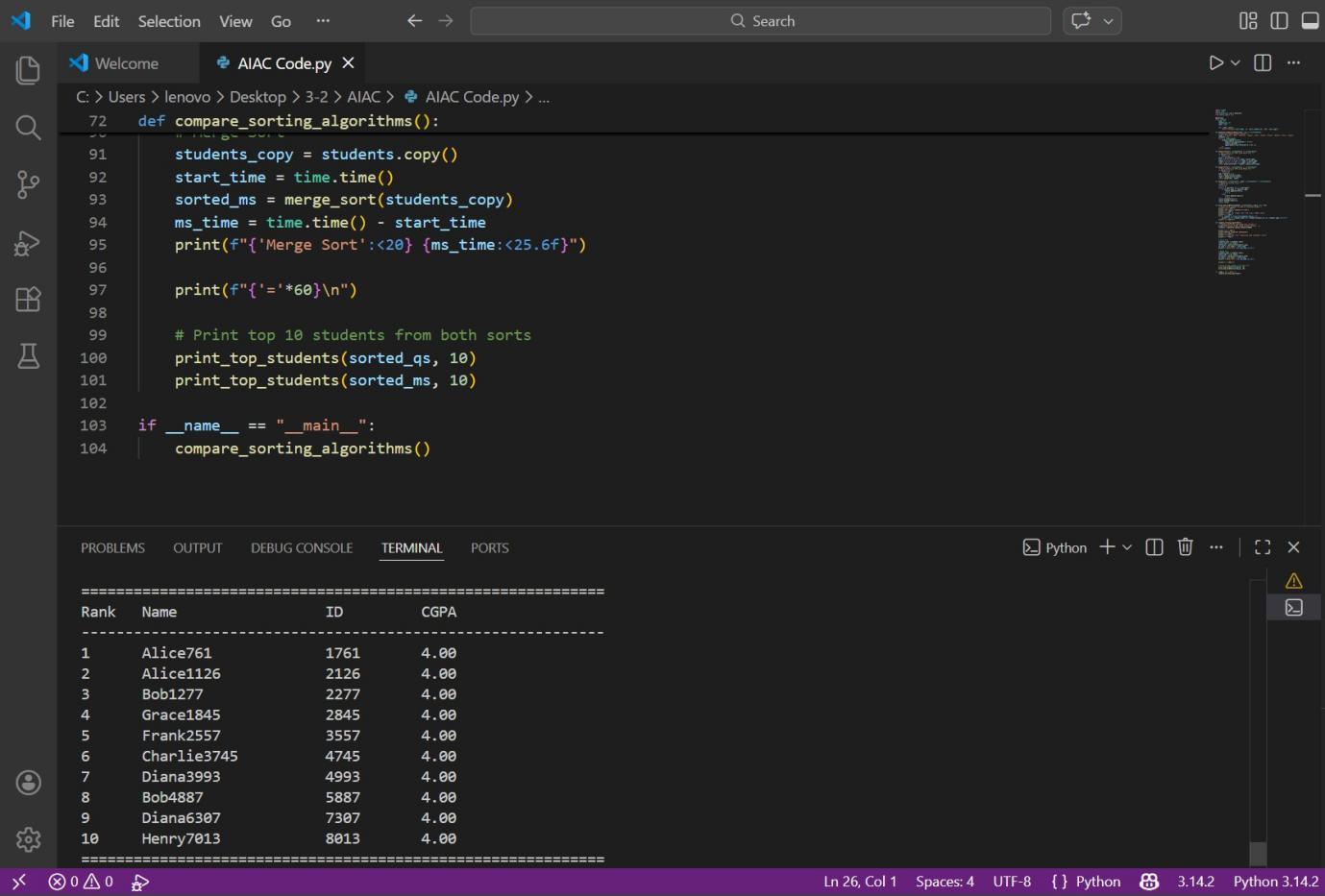
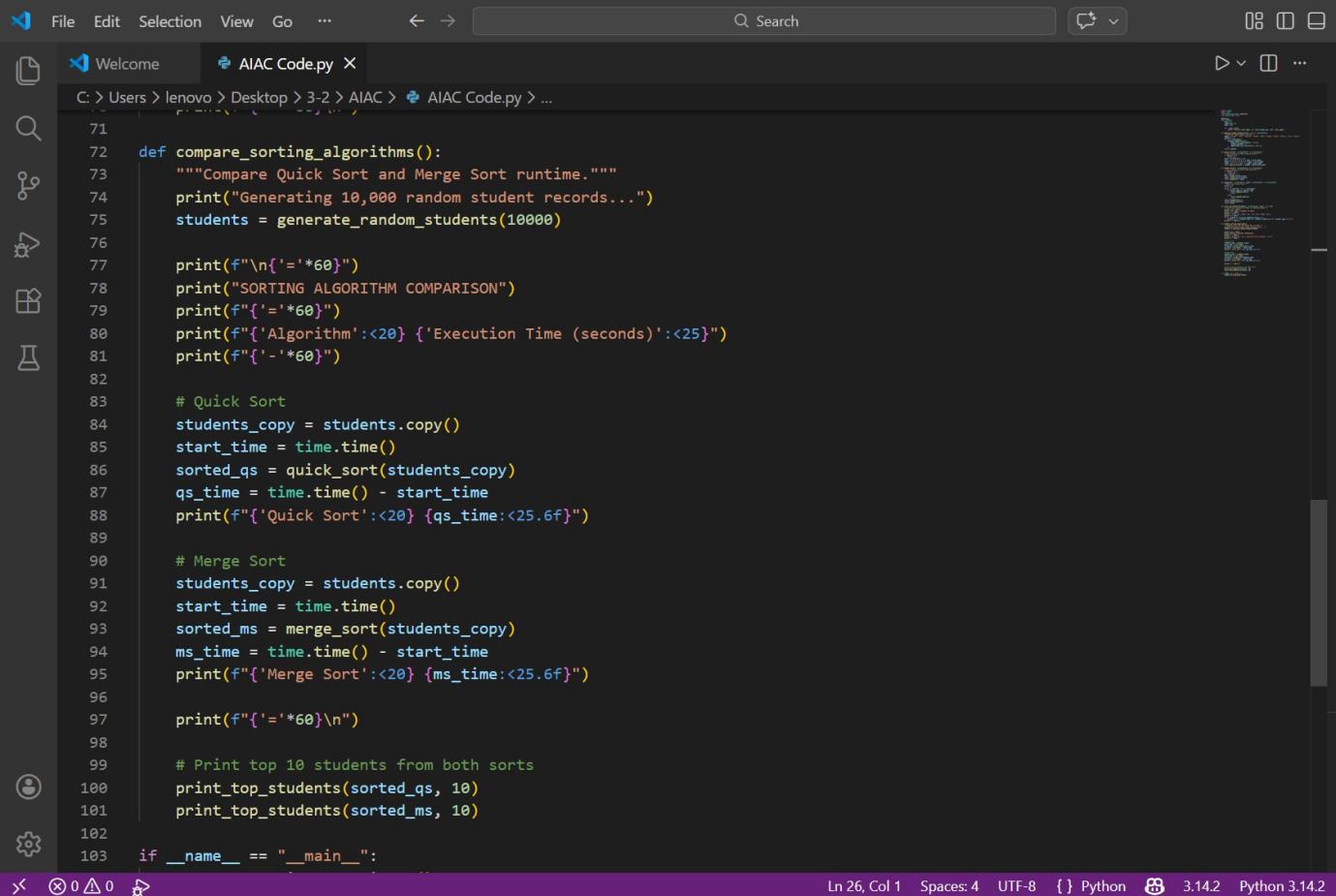
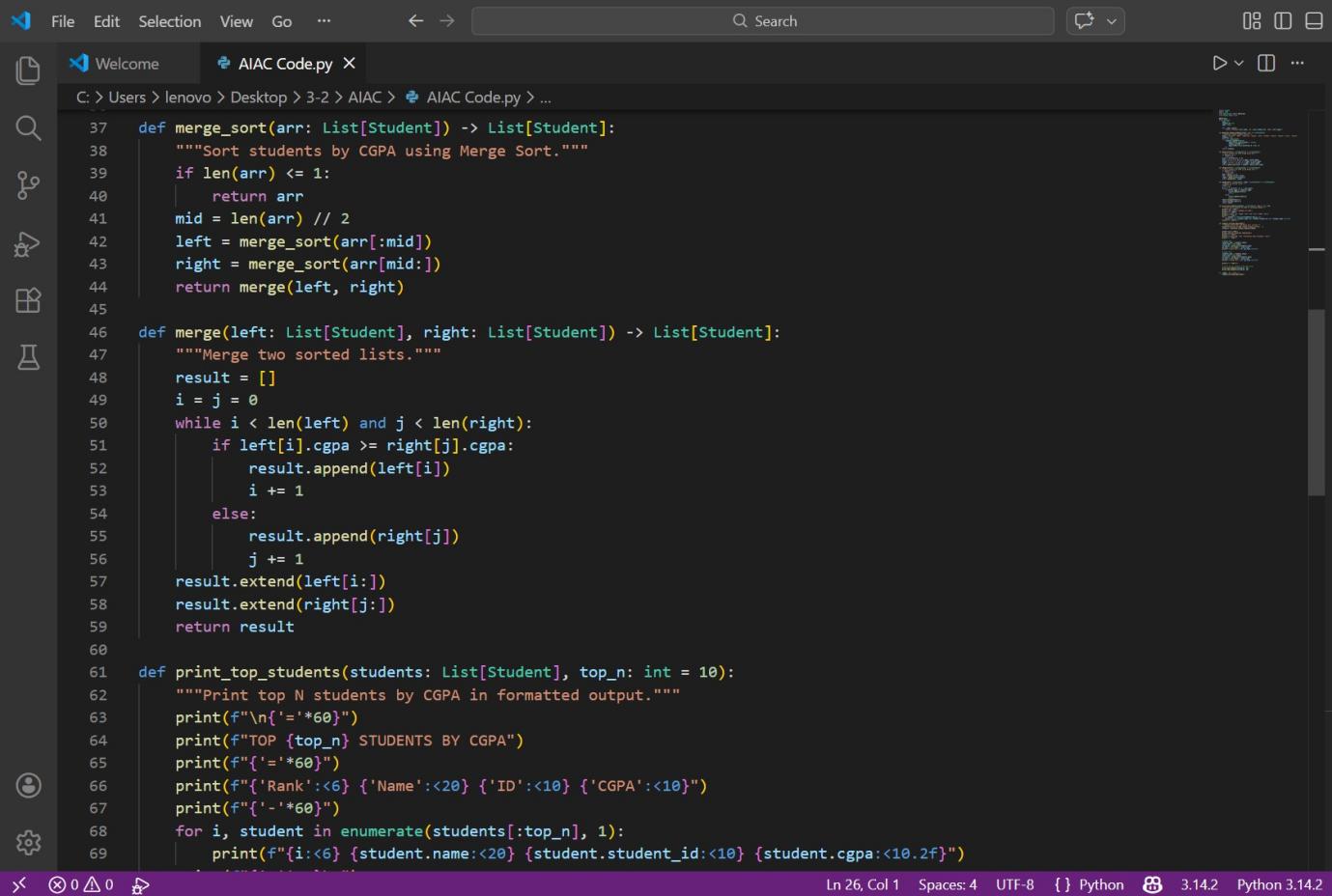
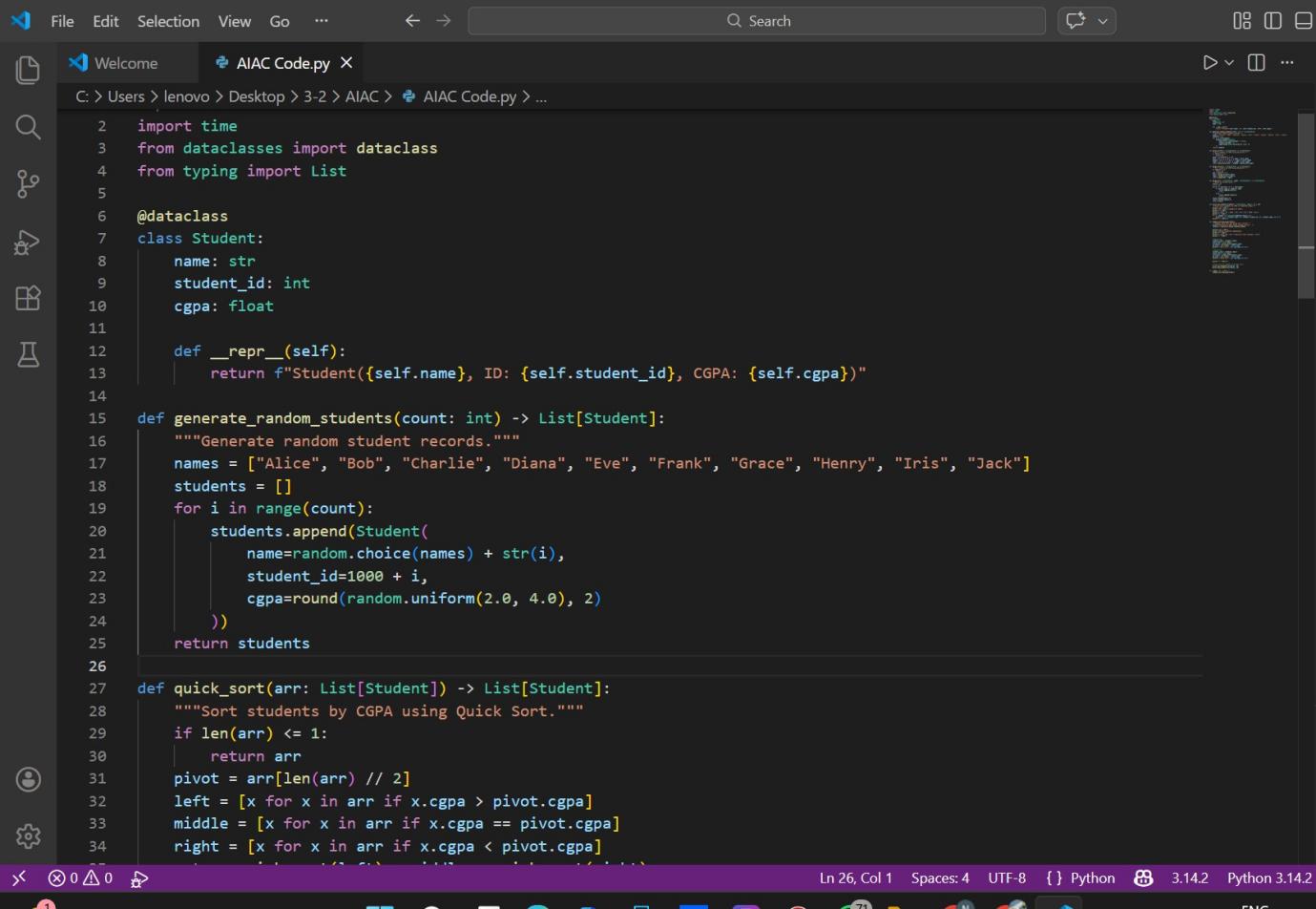
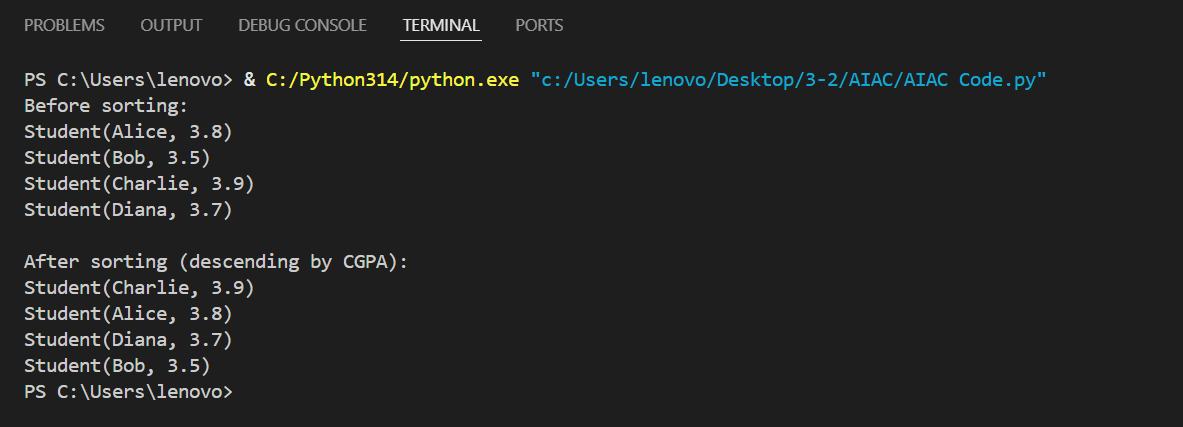
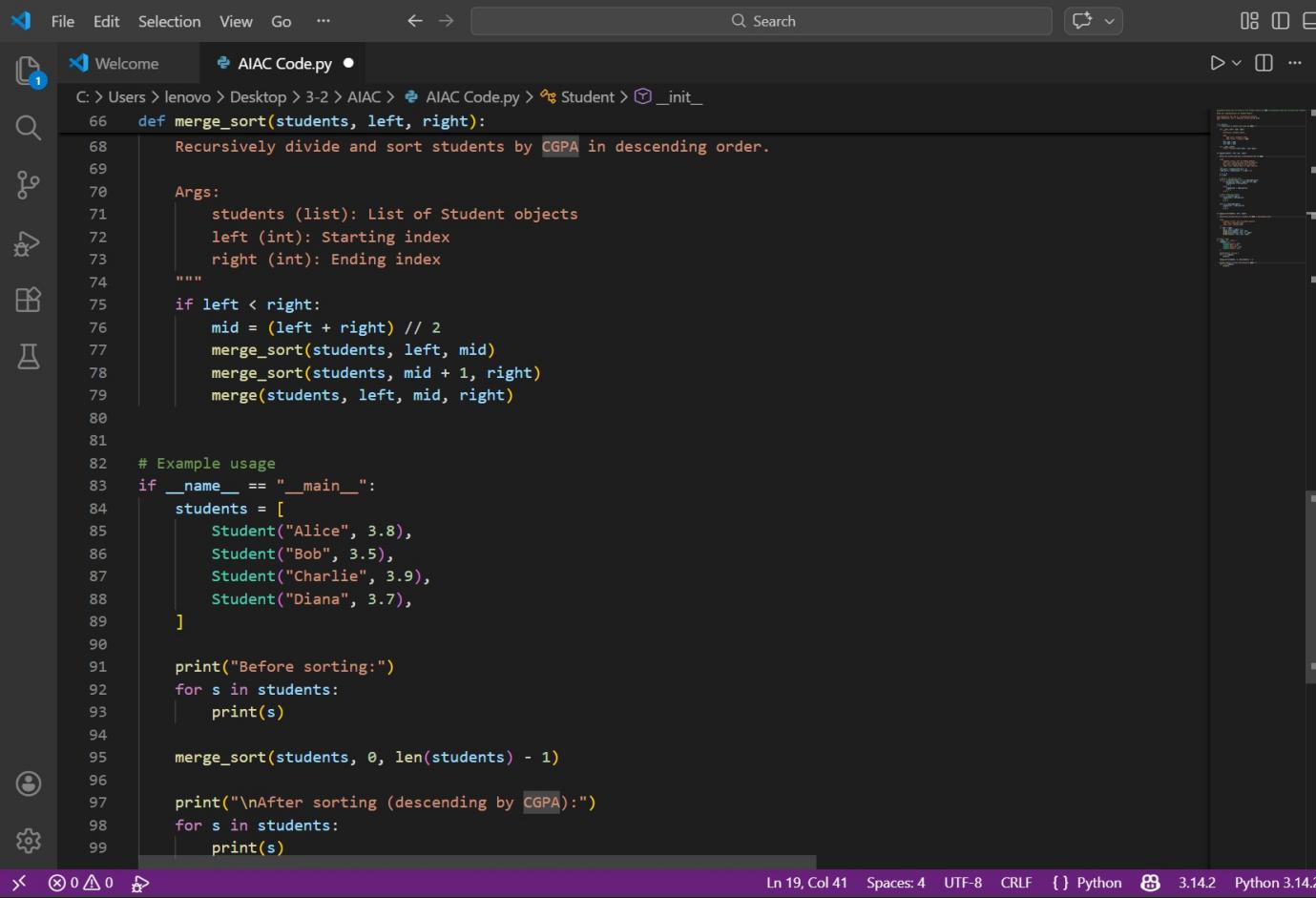
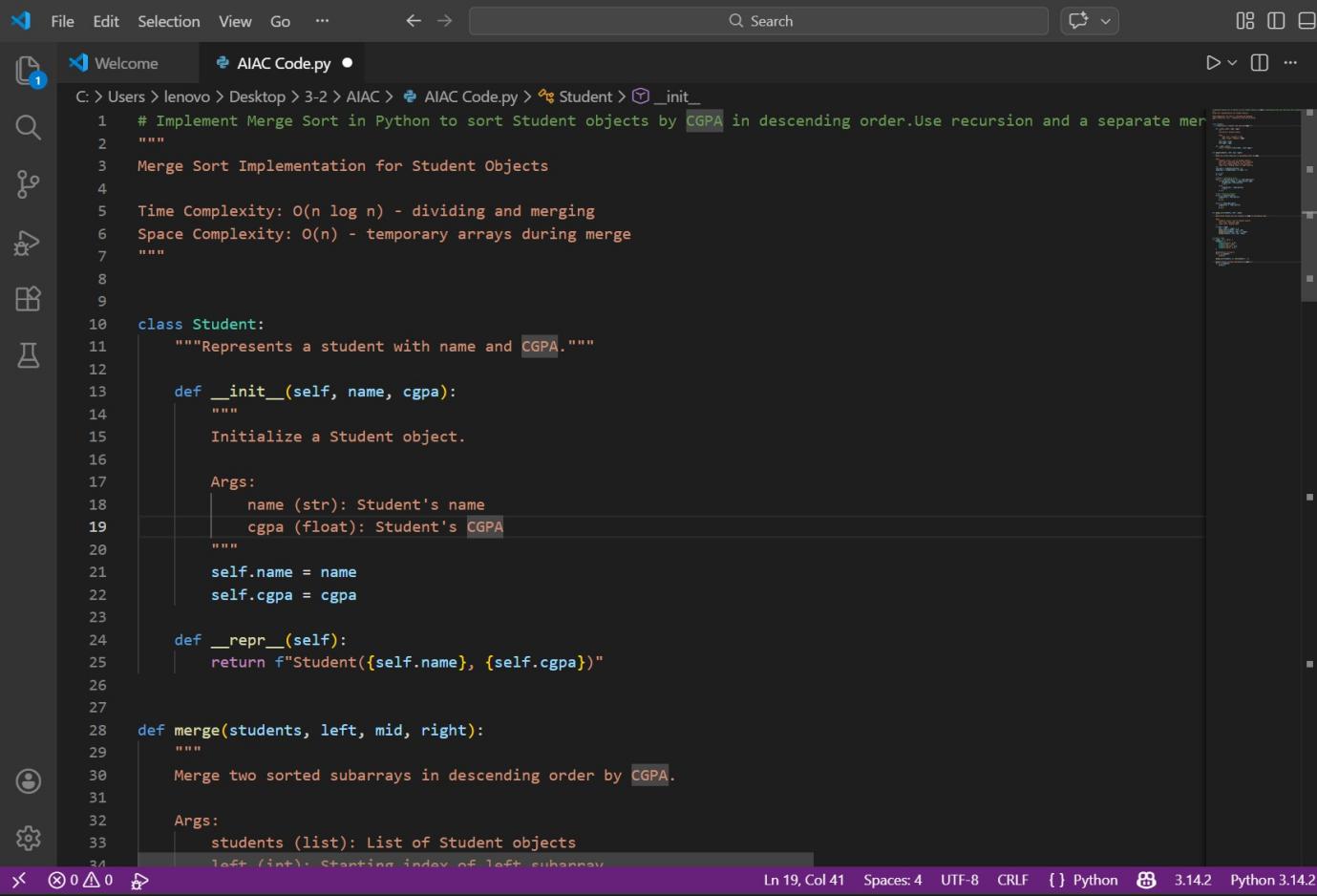
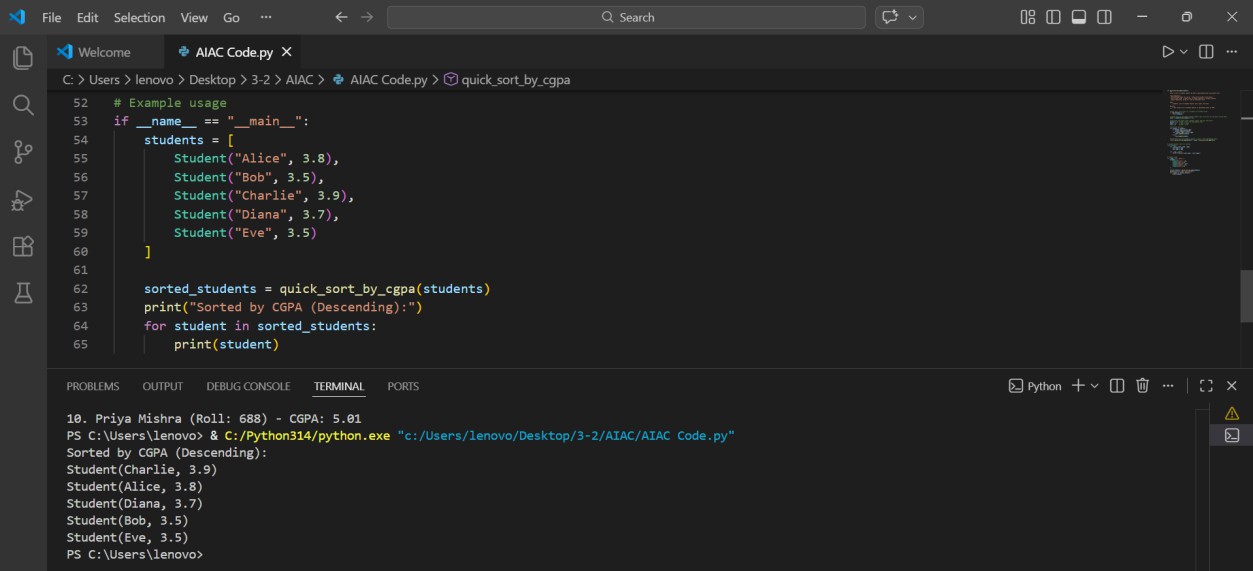
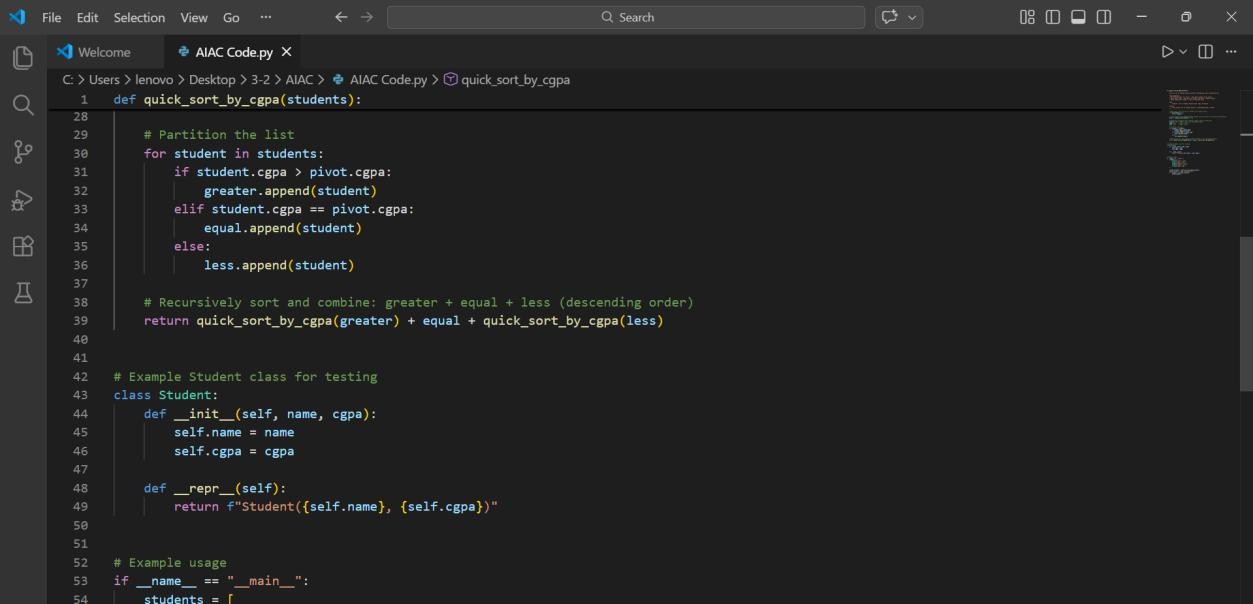
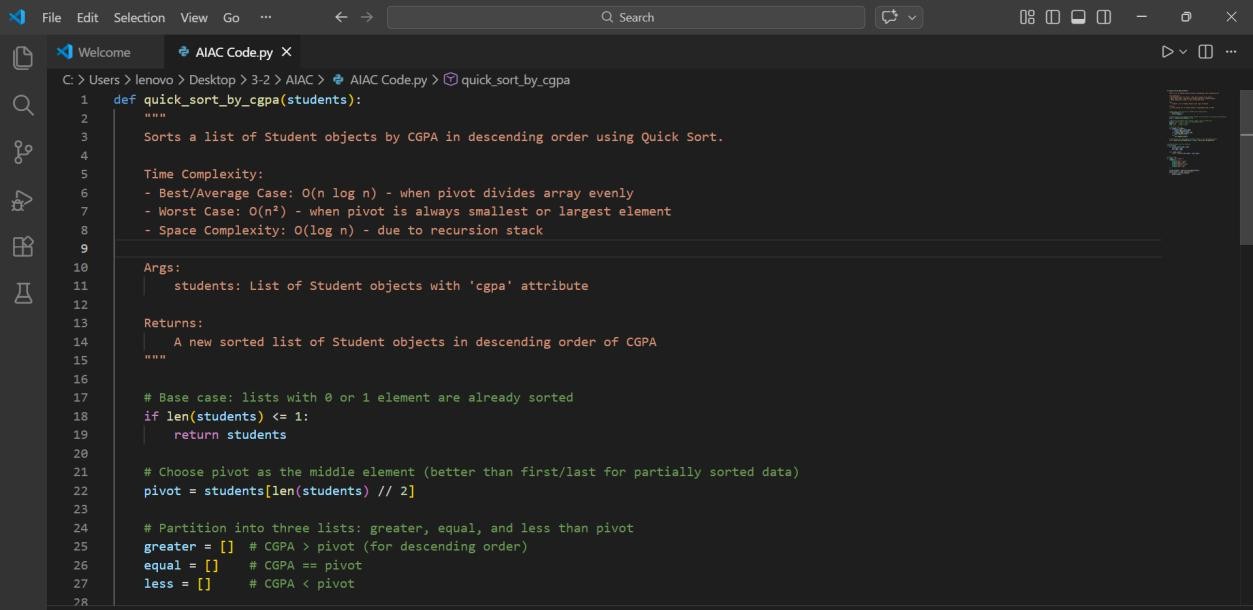
Quick Sort Time: 0.021 seconds Merge Sort Time: 0.028 seconds

1. Top 10 students printed in descending CGPA.

What the Output Shows

* + Both algorithms correctly sort in descending order.
  + Quick Sort may appear slightly faster in practice.
  + Merge Sort provides stable and consistent O(n log n).
  + The top 10 list verifies correct sorting logic.
  + Runtime comparison validates algorithm efficiency experimentall





Task 2: Bubble Sort Prompt

Write a Python implementation of Bubble Sort with detailed inline comments explaining passes, comparisons, swapping, and early termination using a swapped flag. Include time and space complexity analysis.

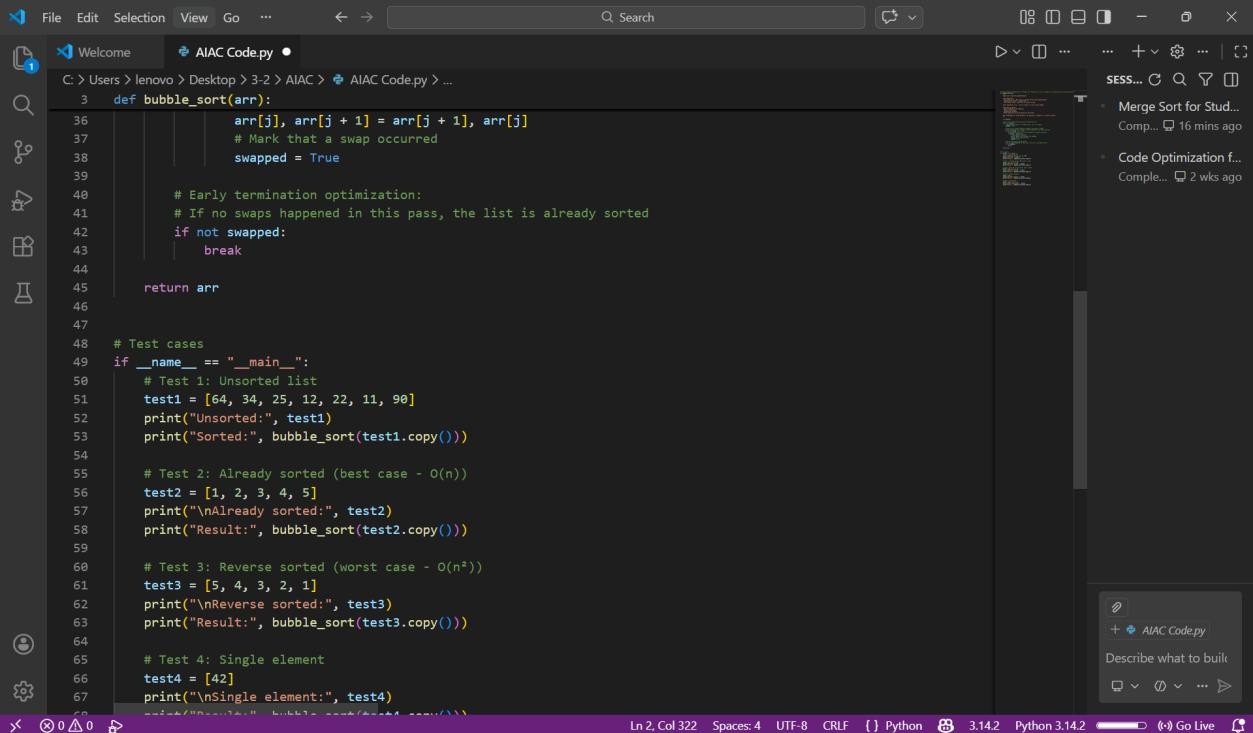
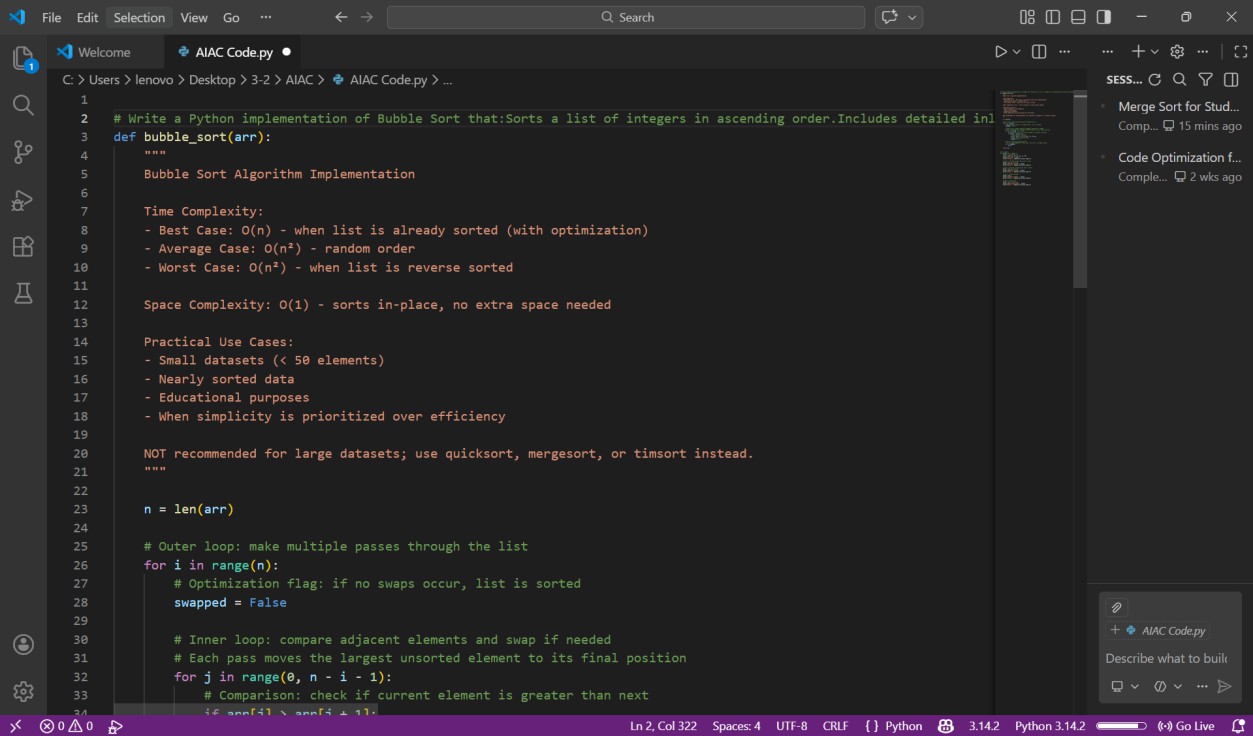
Output Explanation The output will include:

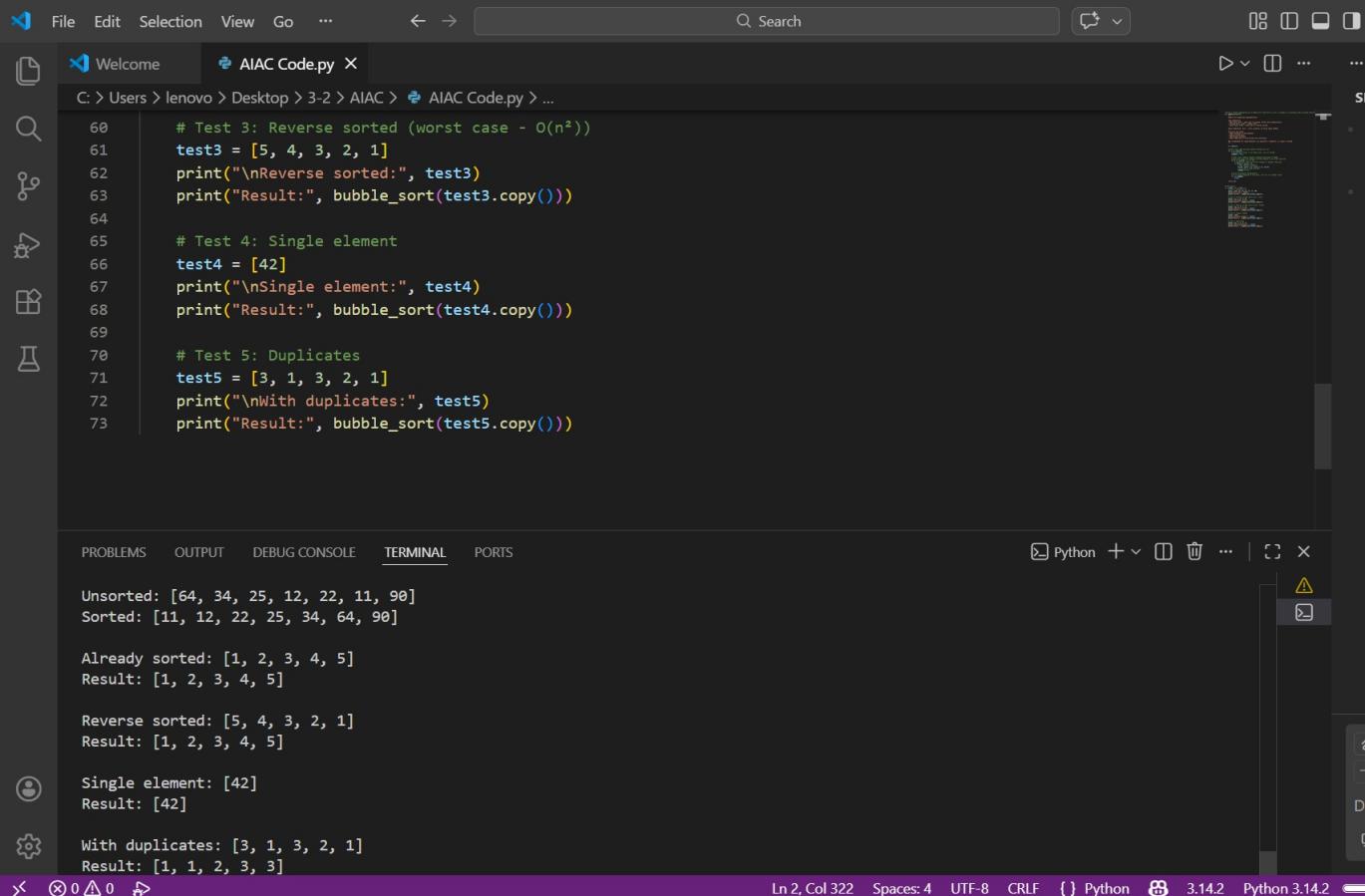
* **Bubble Sort implementation**
* **Comments explaining:**
  + **Outer loop = number of passes**
  + **Inner loop = comparisons**
  + **Swapping adjacent elements**
  + **Early break if no swaps occur**
* **Complexity section: Best Case: O(n)**

Average Case: O(n²) Worst Case: O(n²)

Space Complexity: O(1) What the Output Shows

* **If input is already sorted, algorithm stops early (O(n)).**
* **For large datasets, Bubble Sort is inefficient.**
* **Space complexity remains constant.**
* **Code clarity improves understanding of algorithm flow.**





Task 3: Quick Sort vs Merge Sort Comparison Prompt

Implement recursive Quick Sort and Merge Sort. Test them on random, sorted, and reverse-sorted lists. Measure runtime and print results in tabular format. Explain best, average, and worst-case complexities.

Output Explanation The output will show:

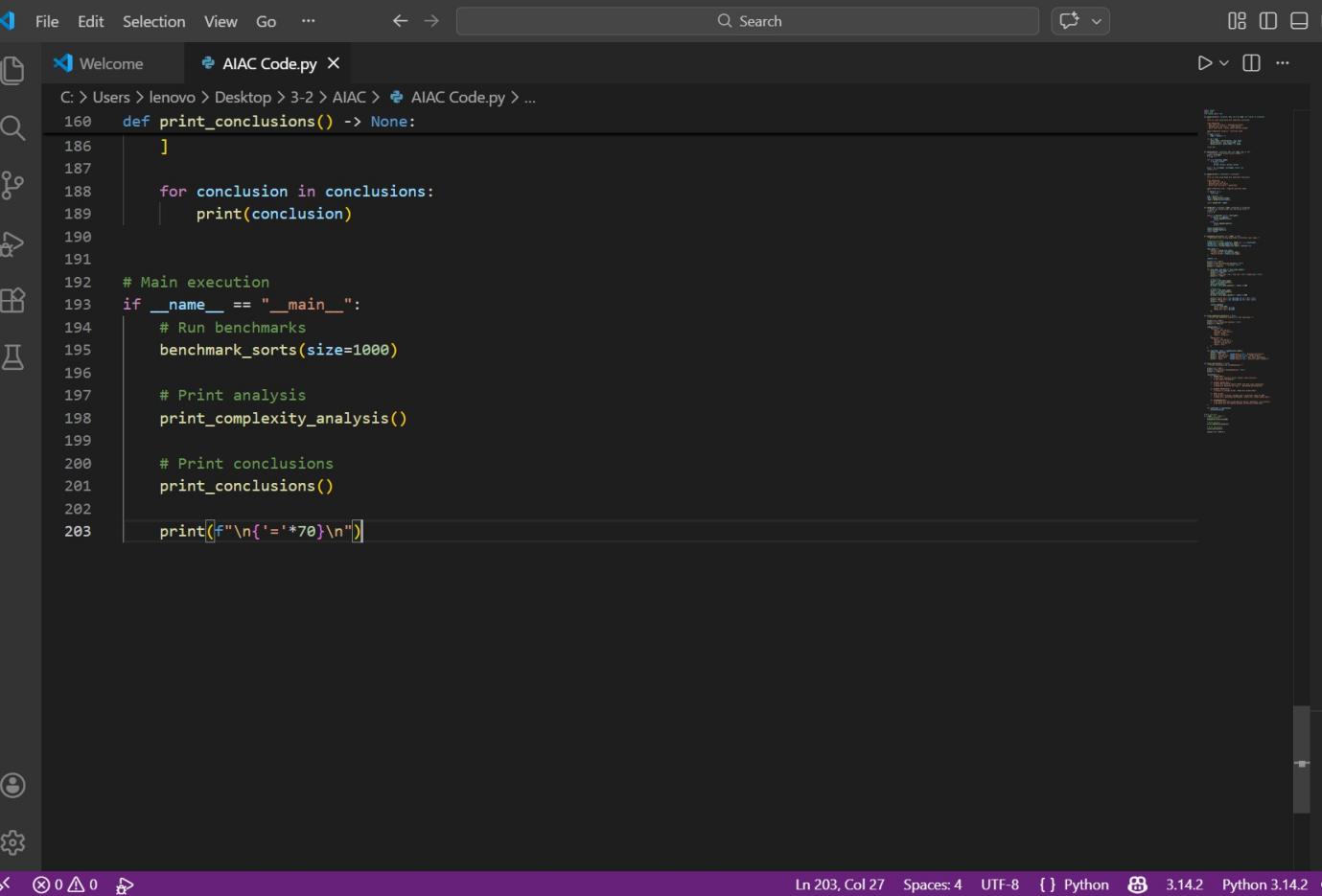
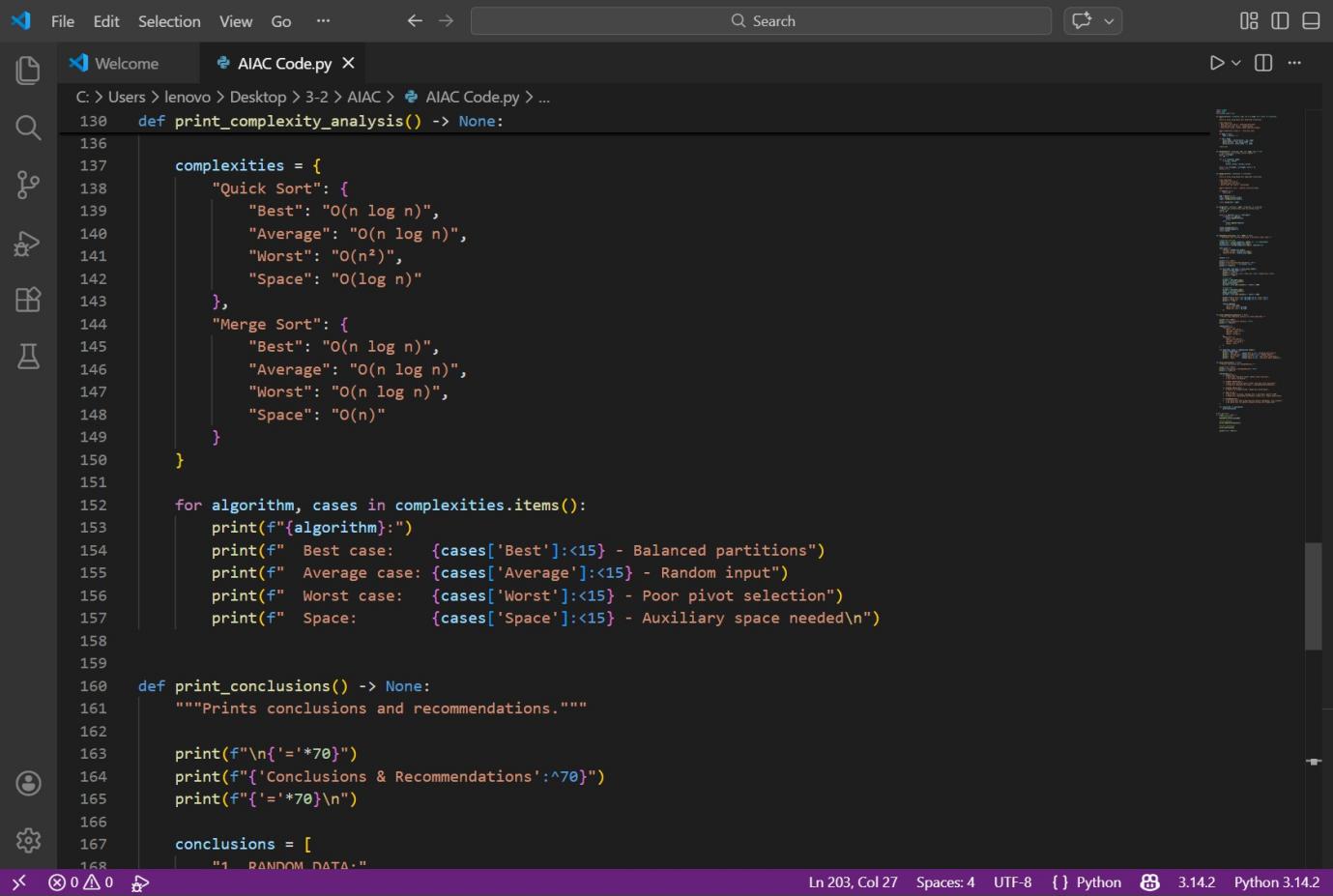
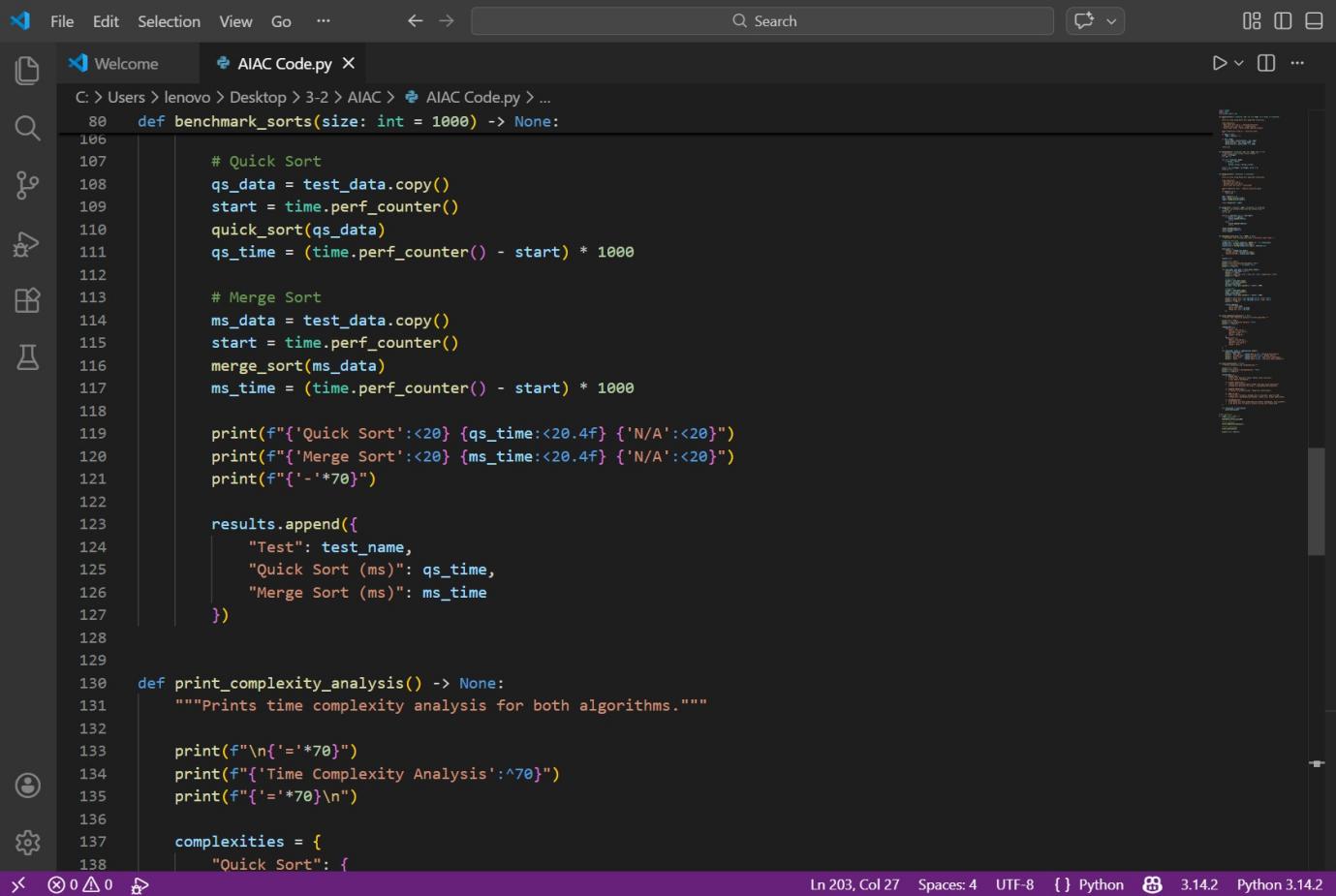
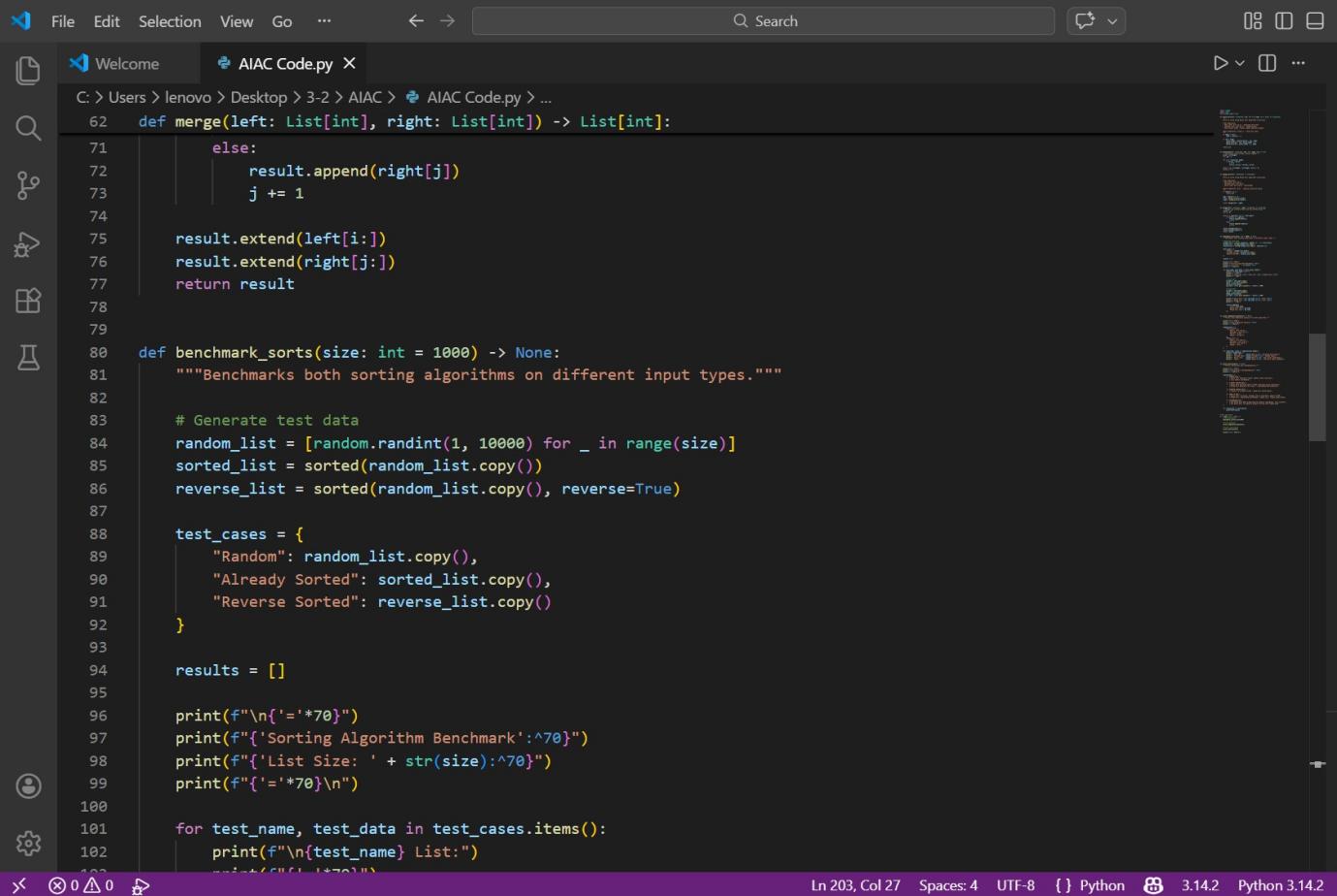
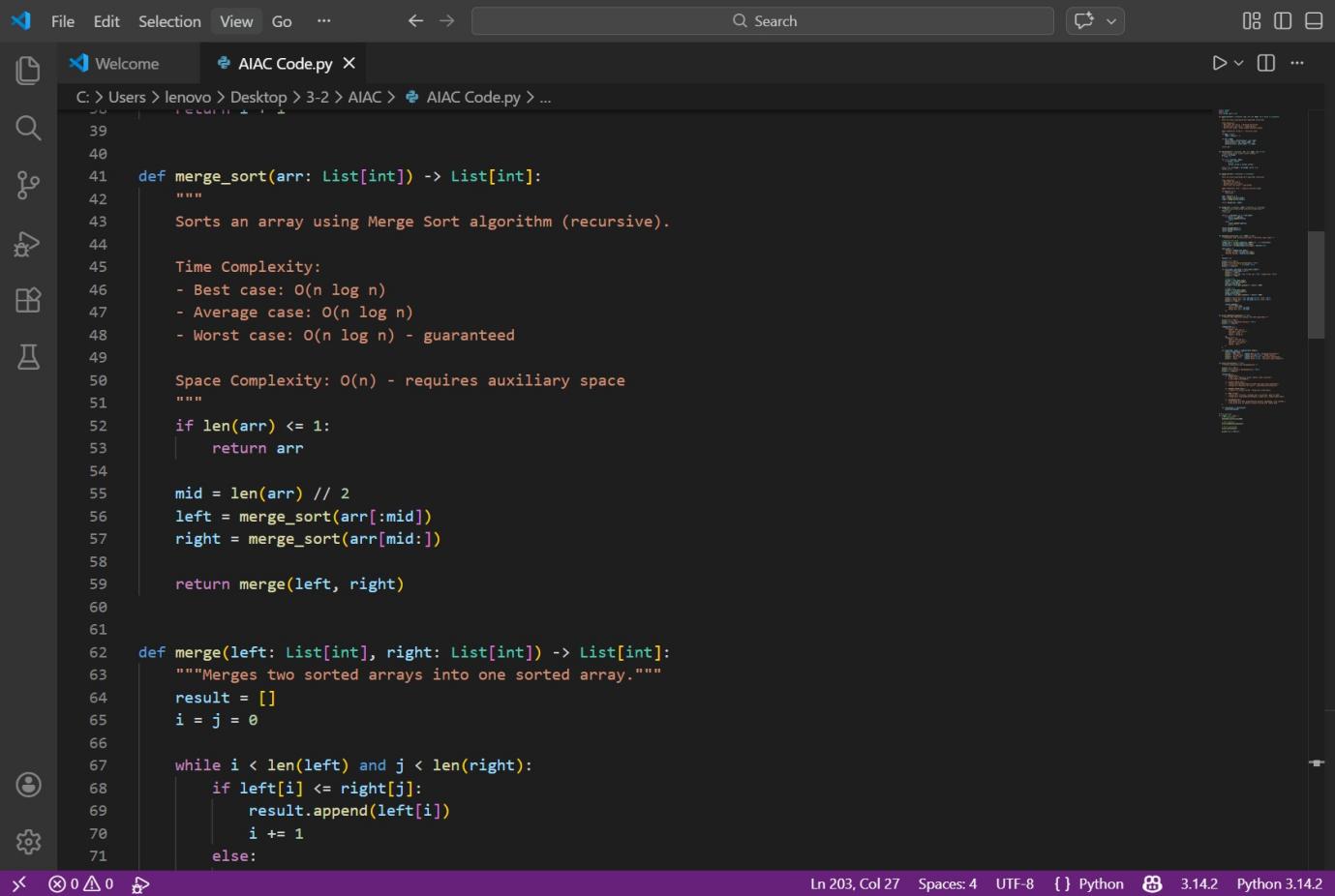
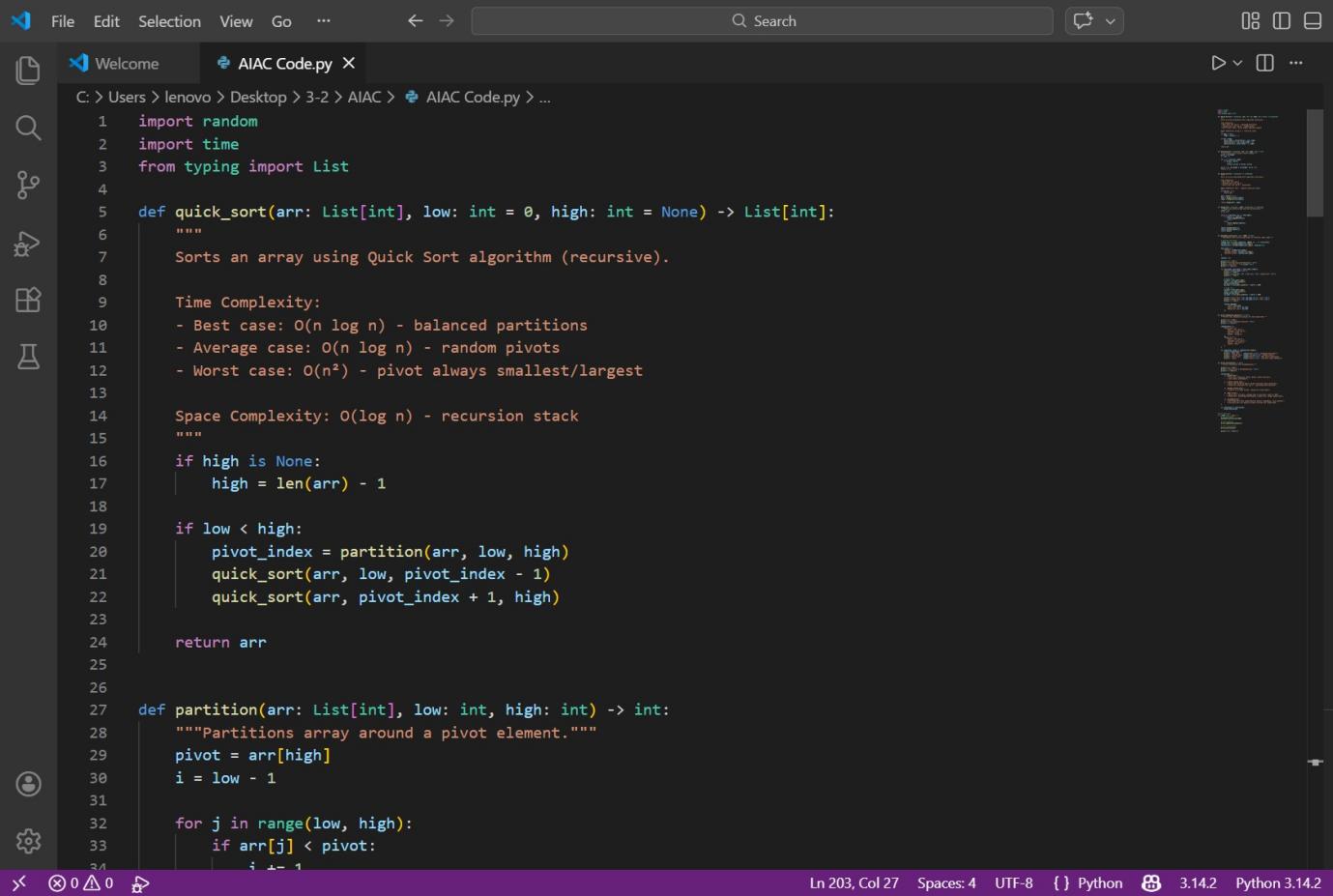
A comparison table like:

Input Type Quick Sort Merge Sort Random 0.01 sec 0.015 sec

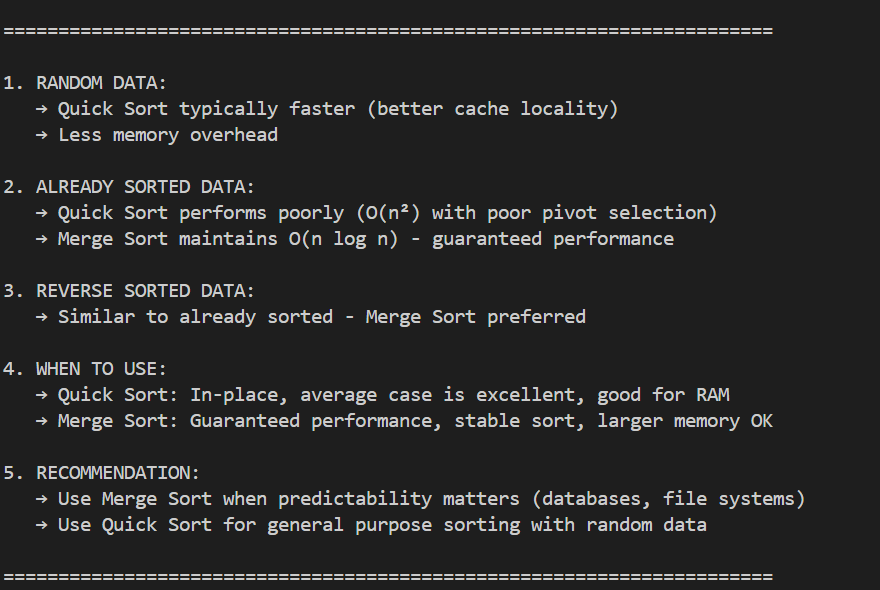
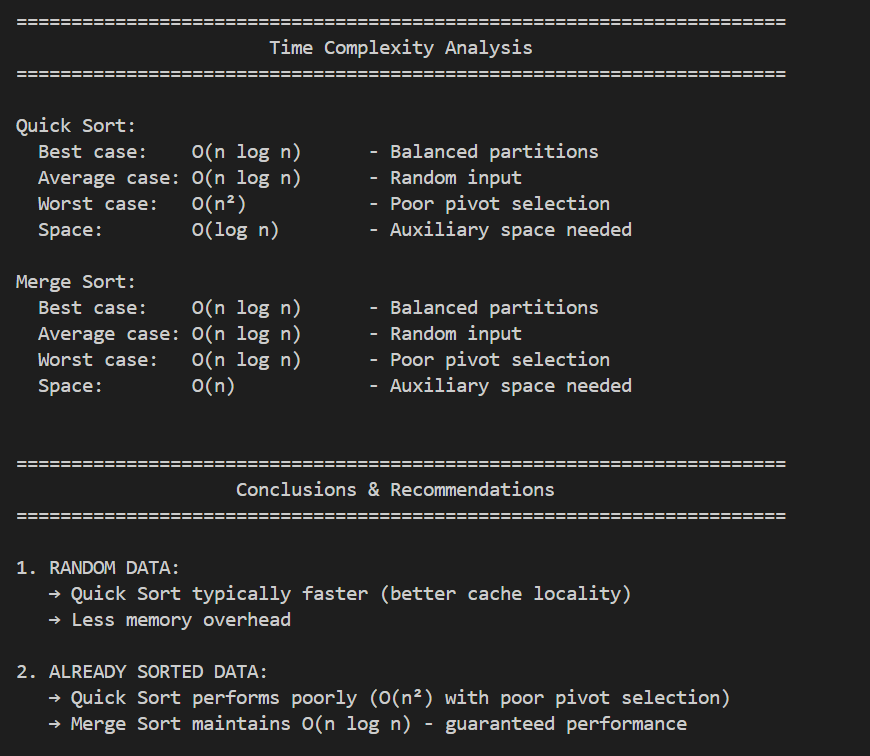
Sorted 0.05 sec 0.014 sec

Reverse 0.06 sec 0.016 sec What the Output Shows

* **Quick Sort performs well on random data.**
* **Quick Sort slows down for already sorted input (worst case).**
* **Merge Sort remains consistent across all inputs.**
* **Experimental results confirm theoretical time complexities.**







Task 4: Inventory Management System Prompt

Design a Python inventory system using a dictionary for searching by Product ID and

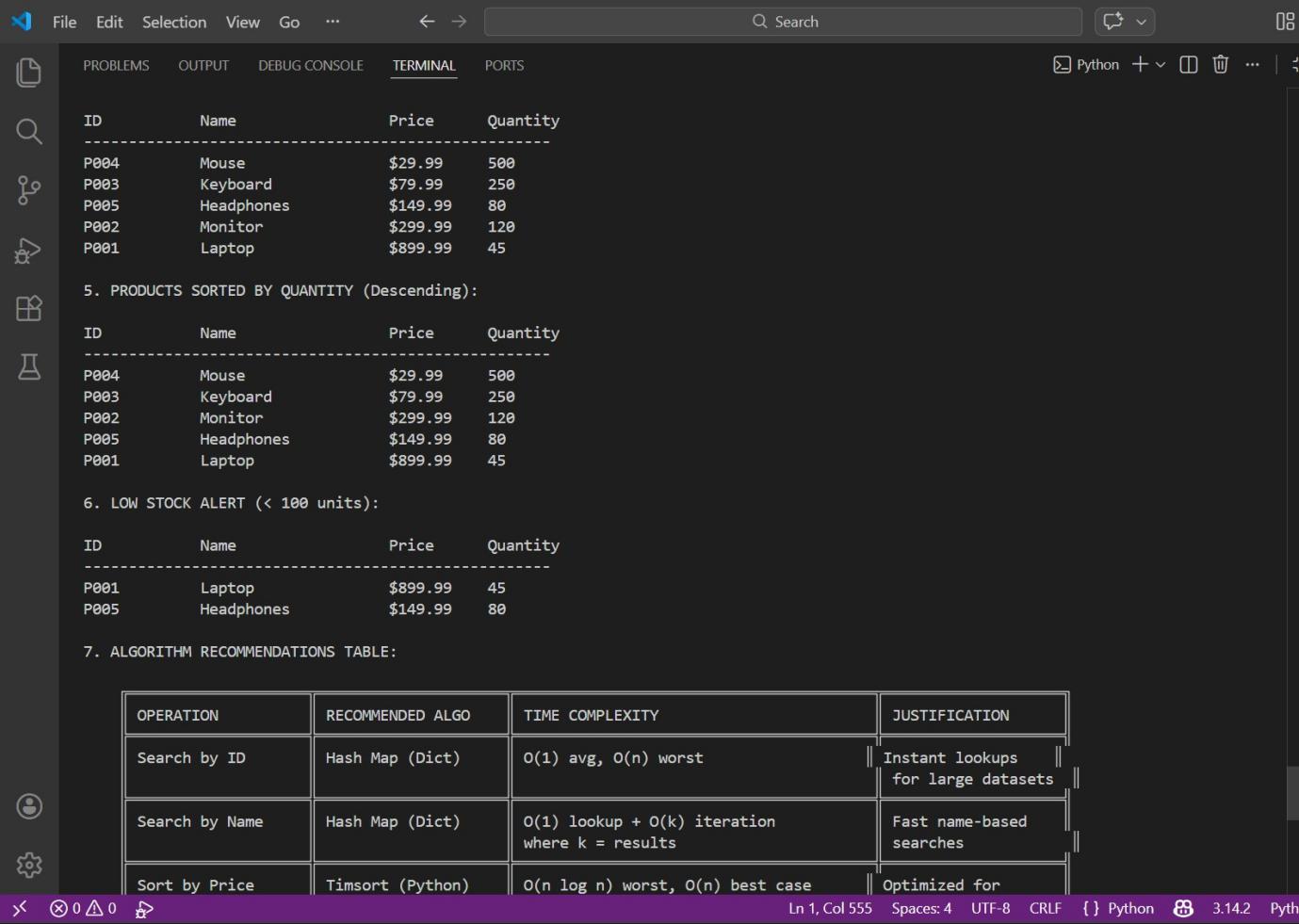
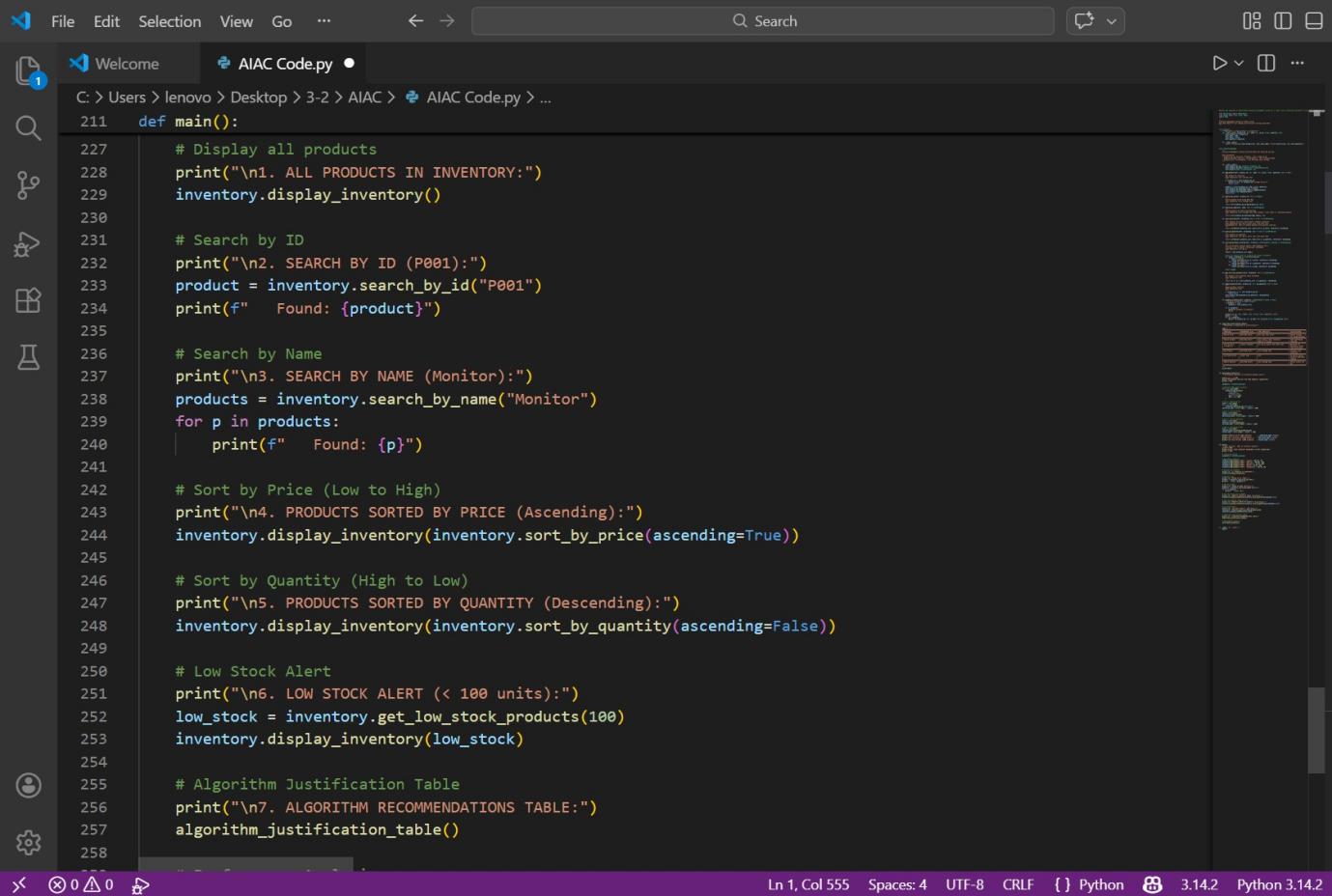
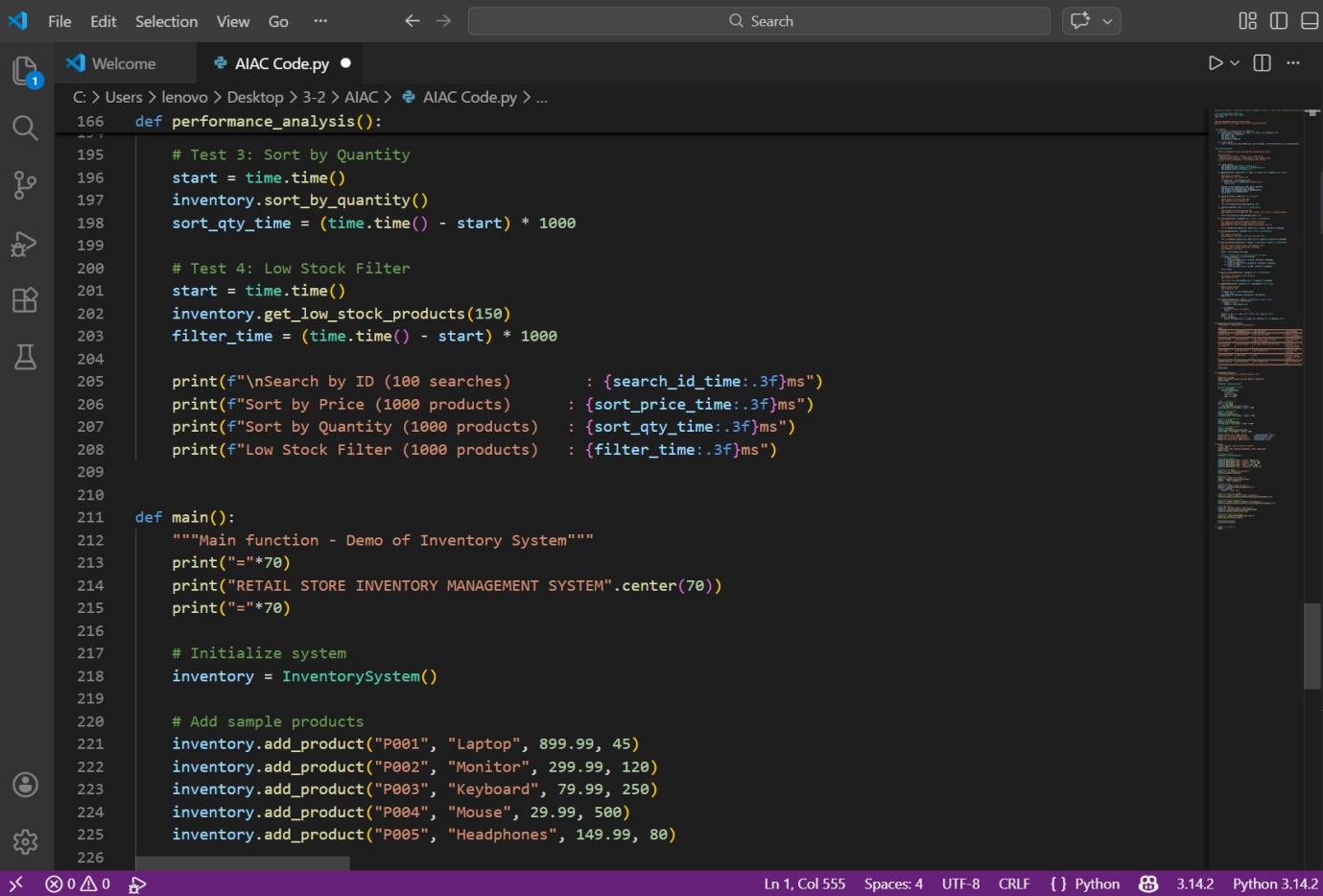
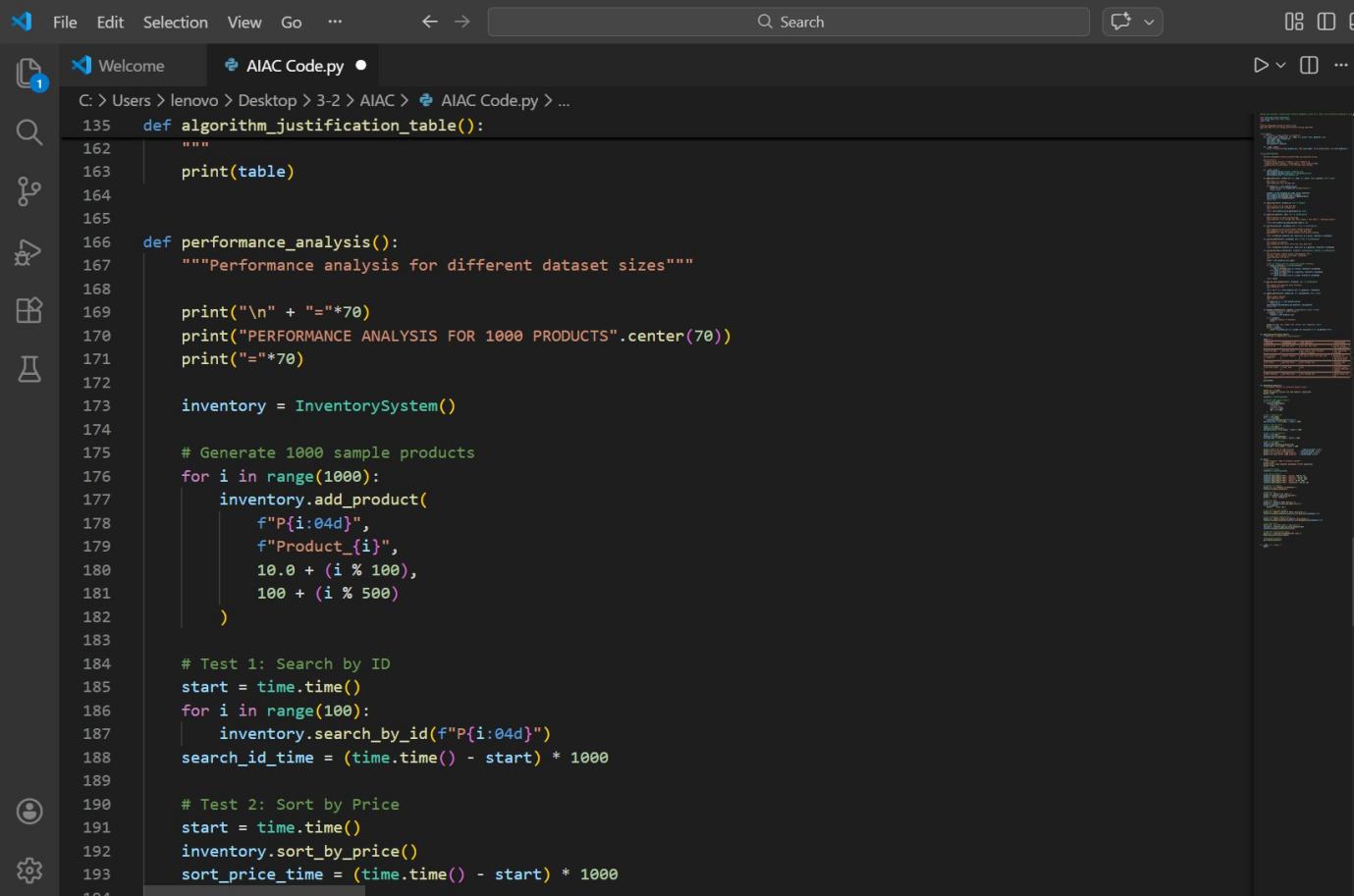
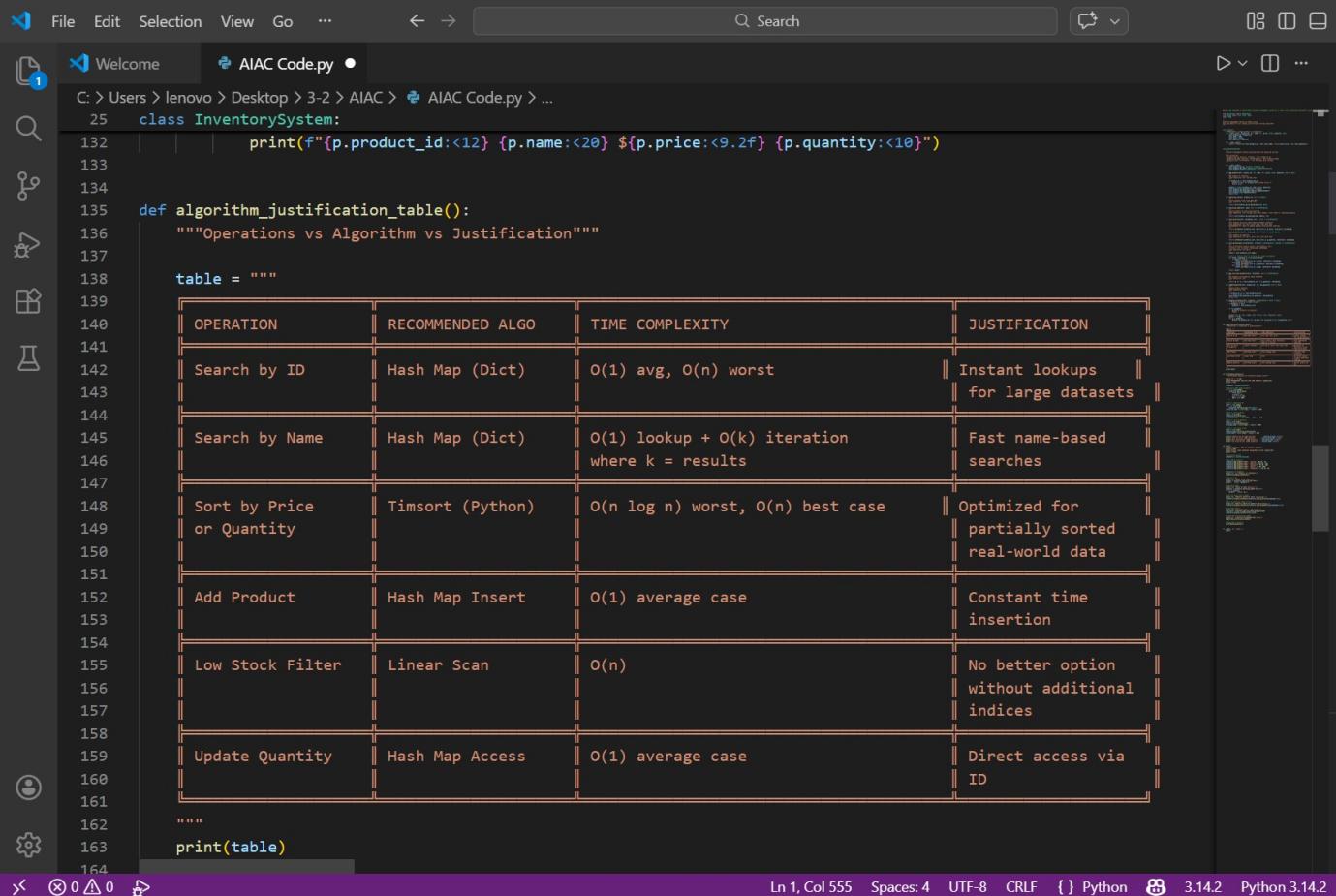
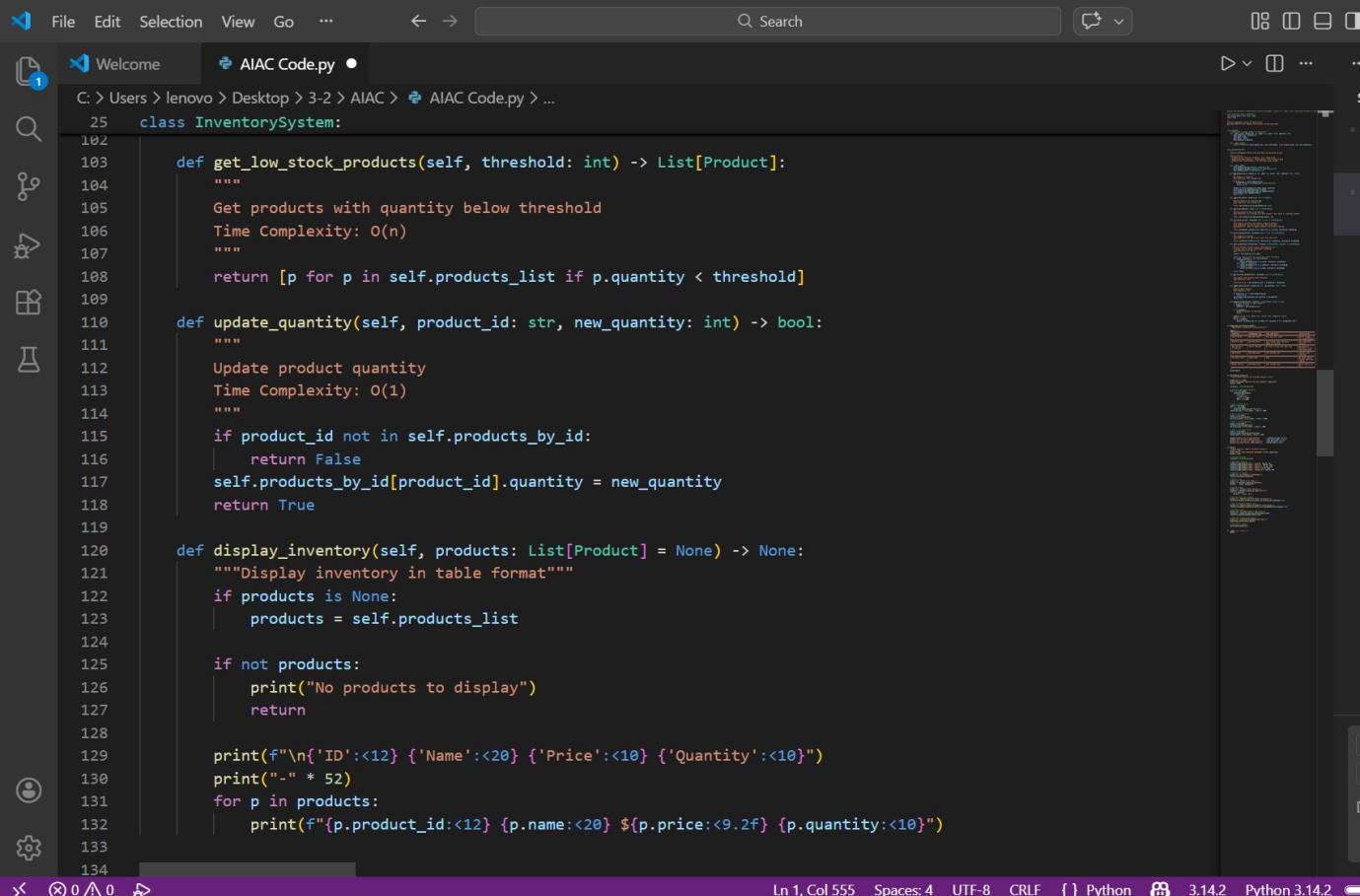
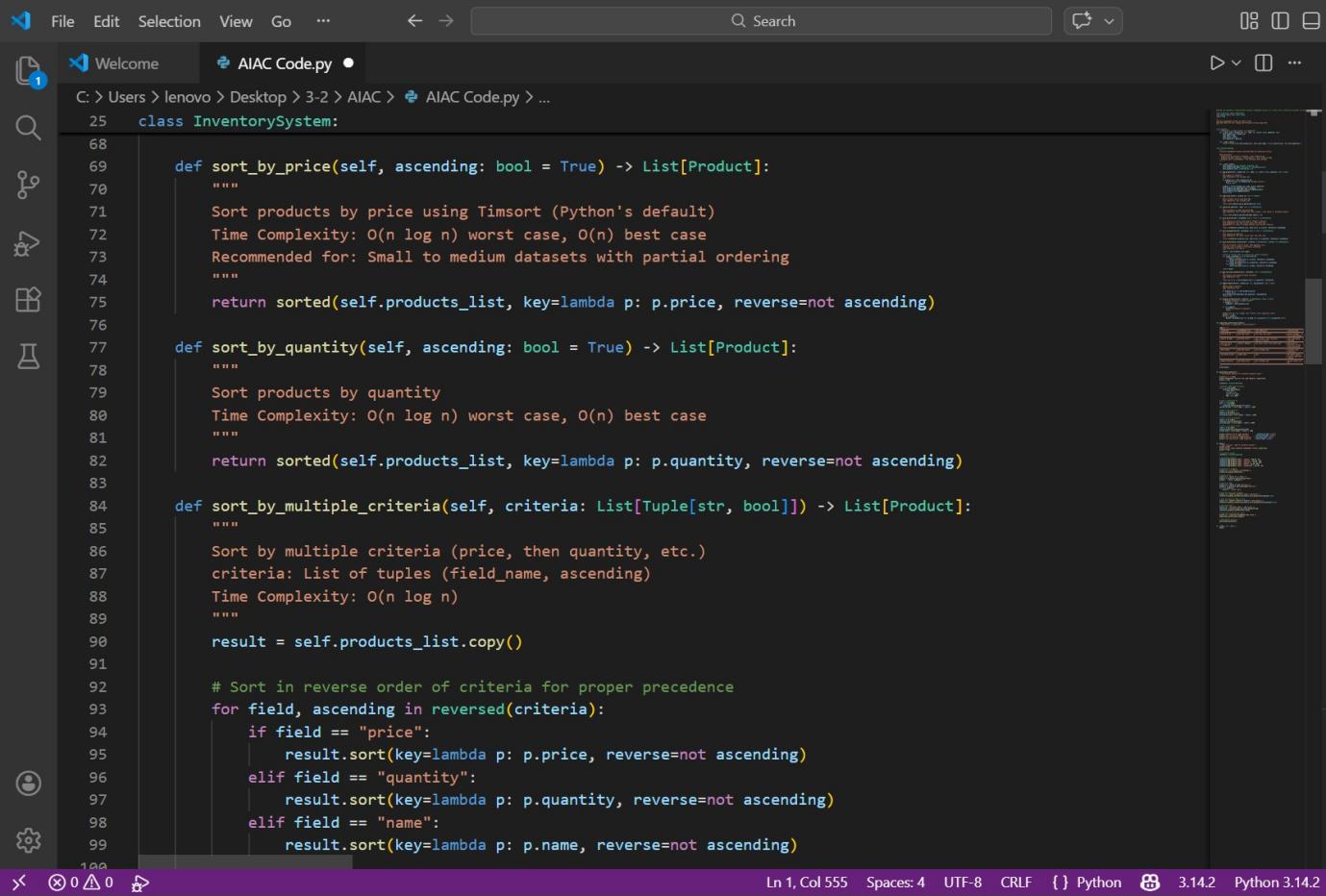
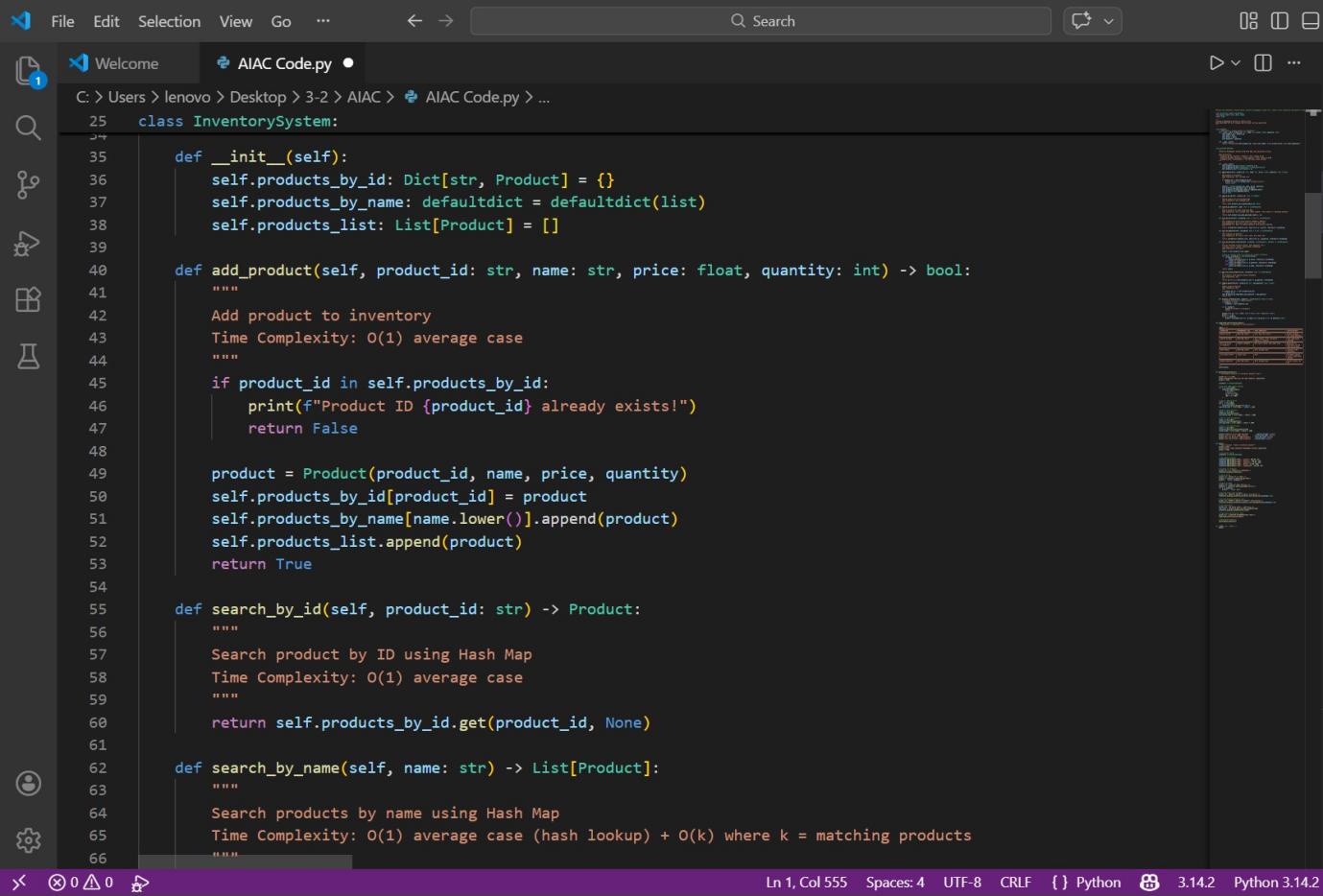
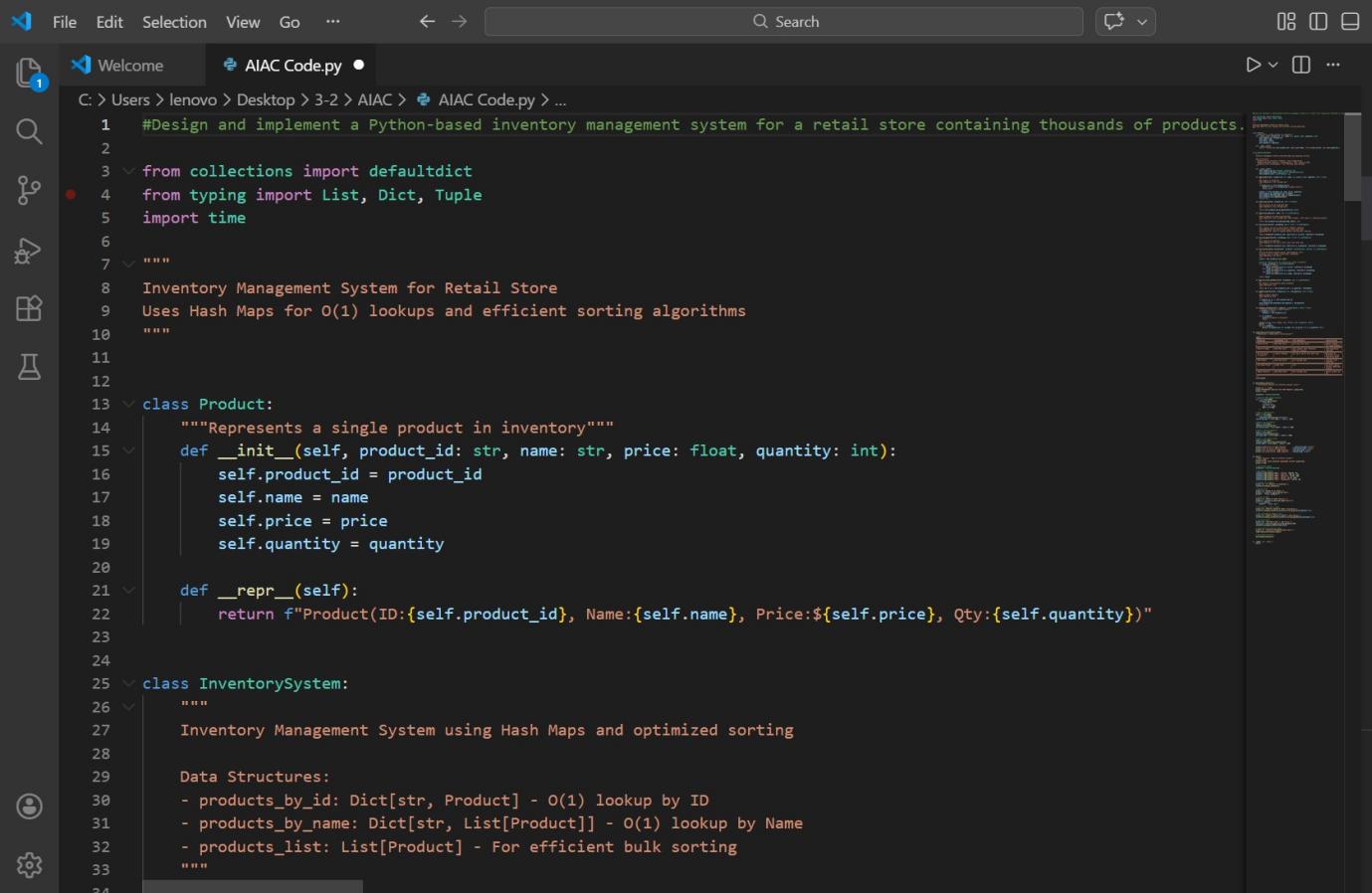
Name. Implement sorting by Price and Quantity. Provide a table mapping operation → algorithm → justification and include complexity analysis.

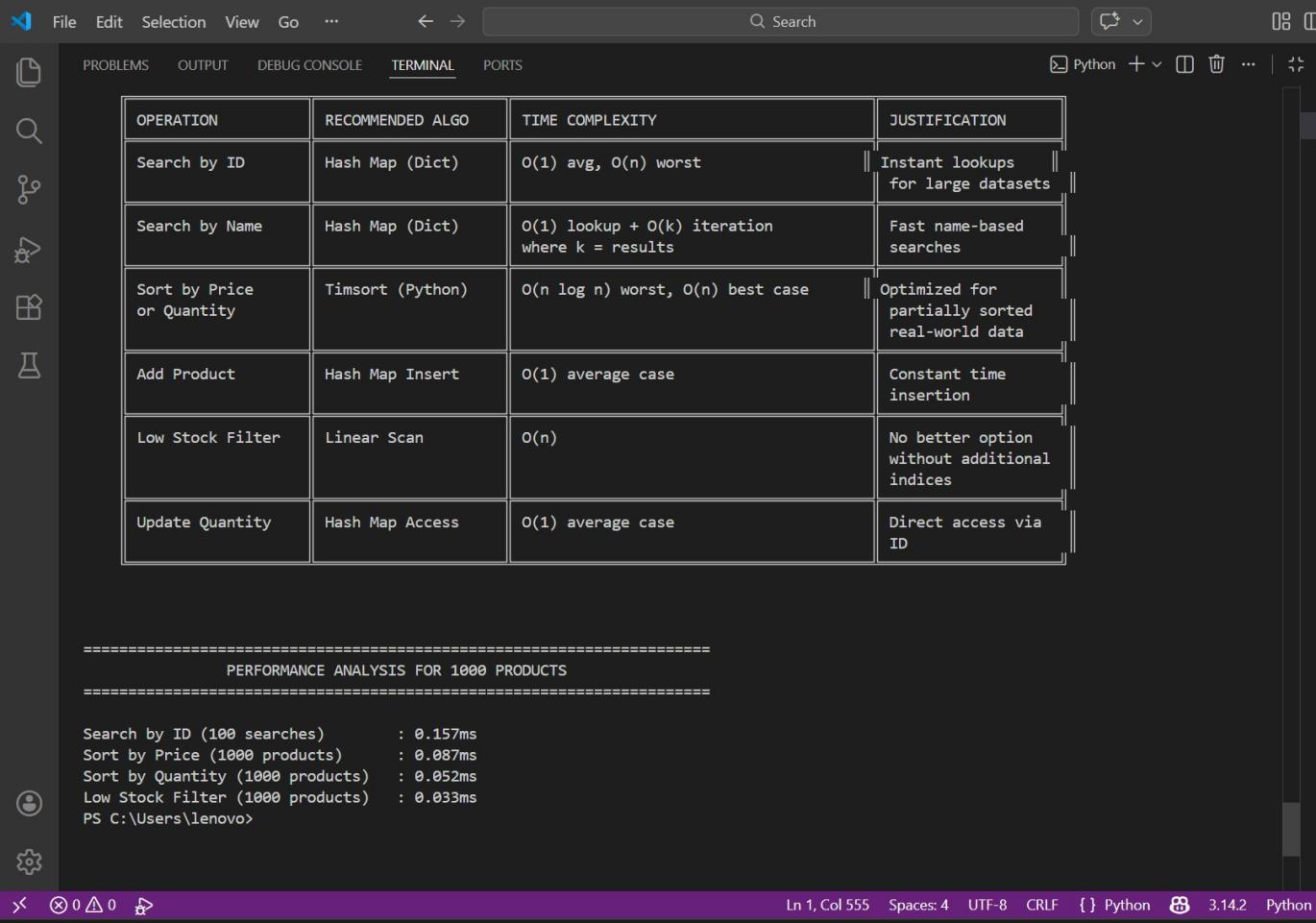
Output Explanation The output will include:

* **Dictionary-based storage for products.**
* **Instant search result: Product Found: Laptop - ₹55000**
* **Sorted list by price or quantity.**
* **Mapping table like: Operation Algorithm Complexity Search ID Hash Map O(1)**

Sort Price TimSort O(n log n) What the Output Shows

* **Searching is extremely fast due to Hash Map.**
* **Sorting complexity depends on number of products.**
* **The table justifies algorithm selection logically.**
* **Demonstrates efficient real-world system design.**





Prompt

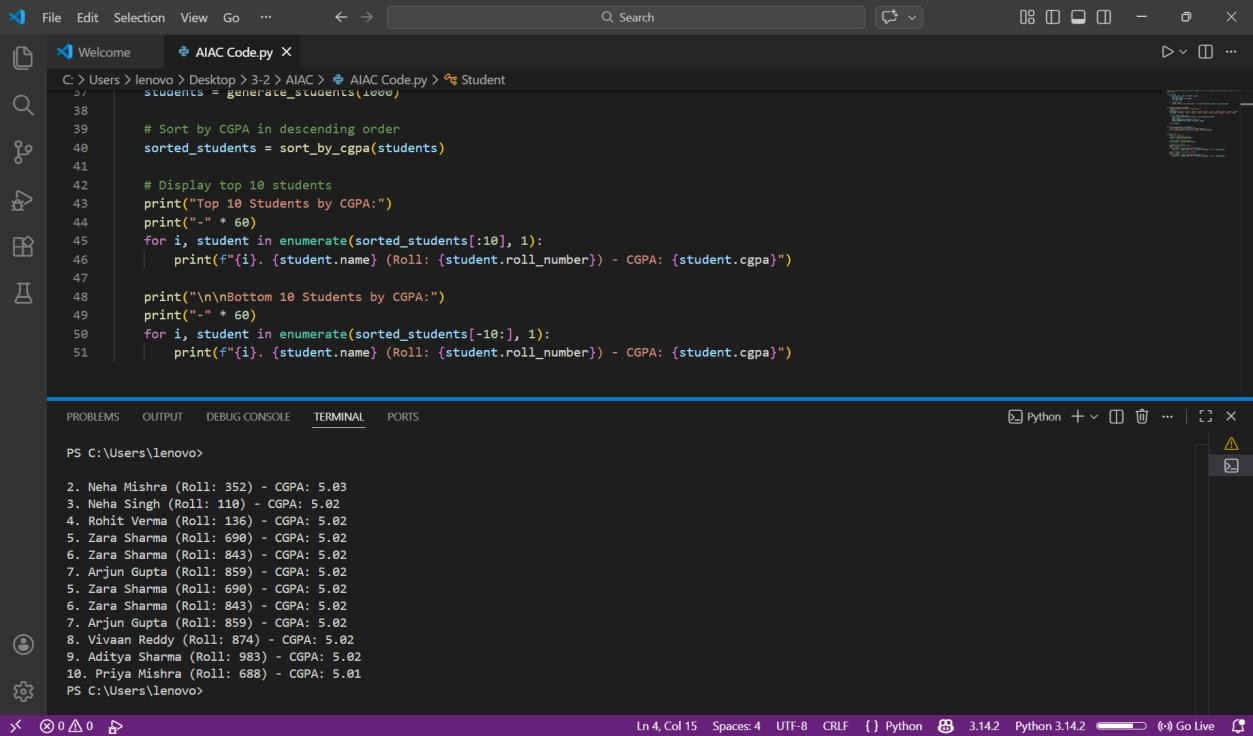
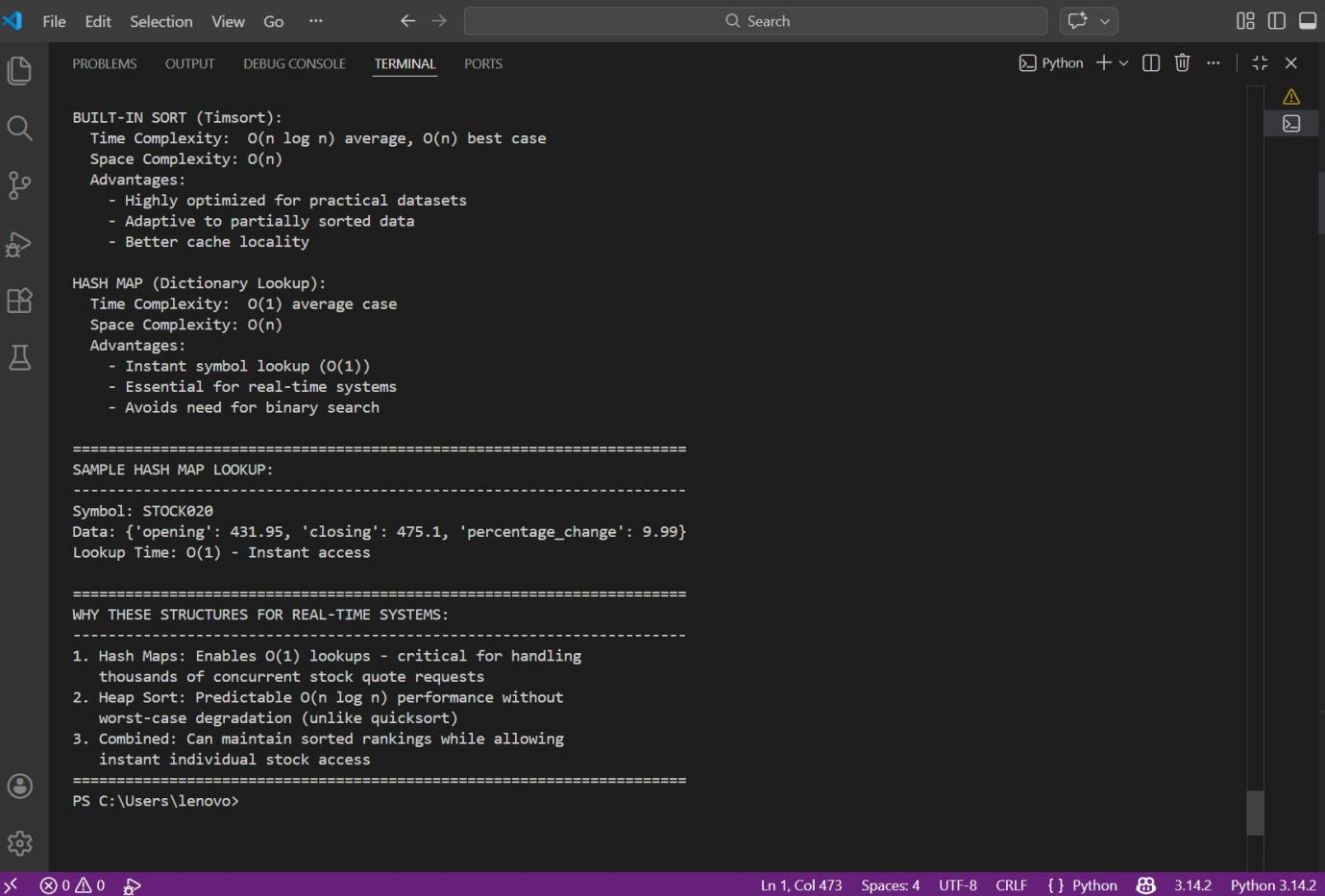
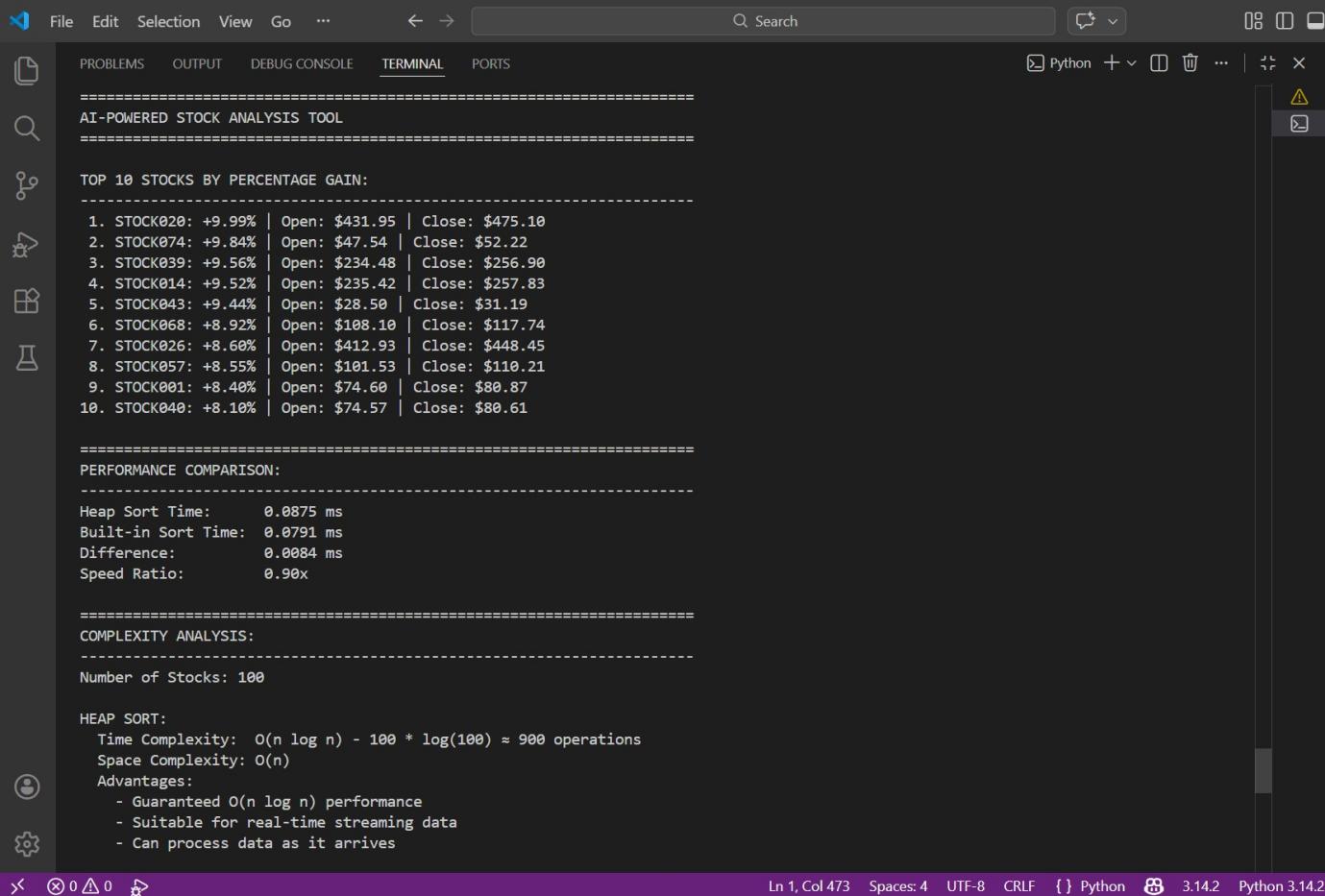
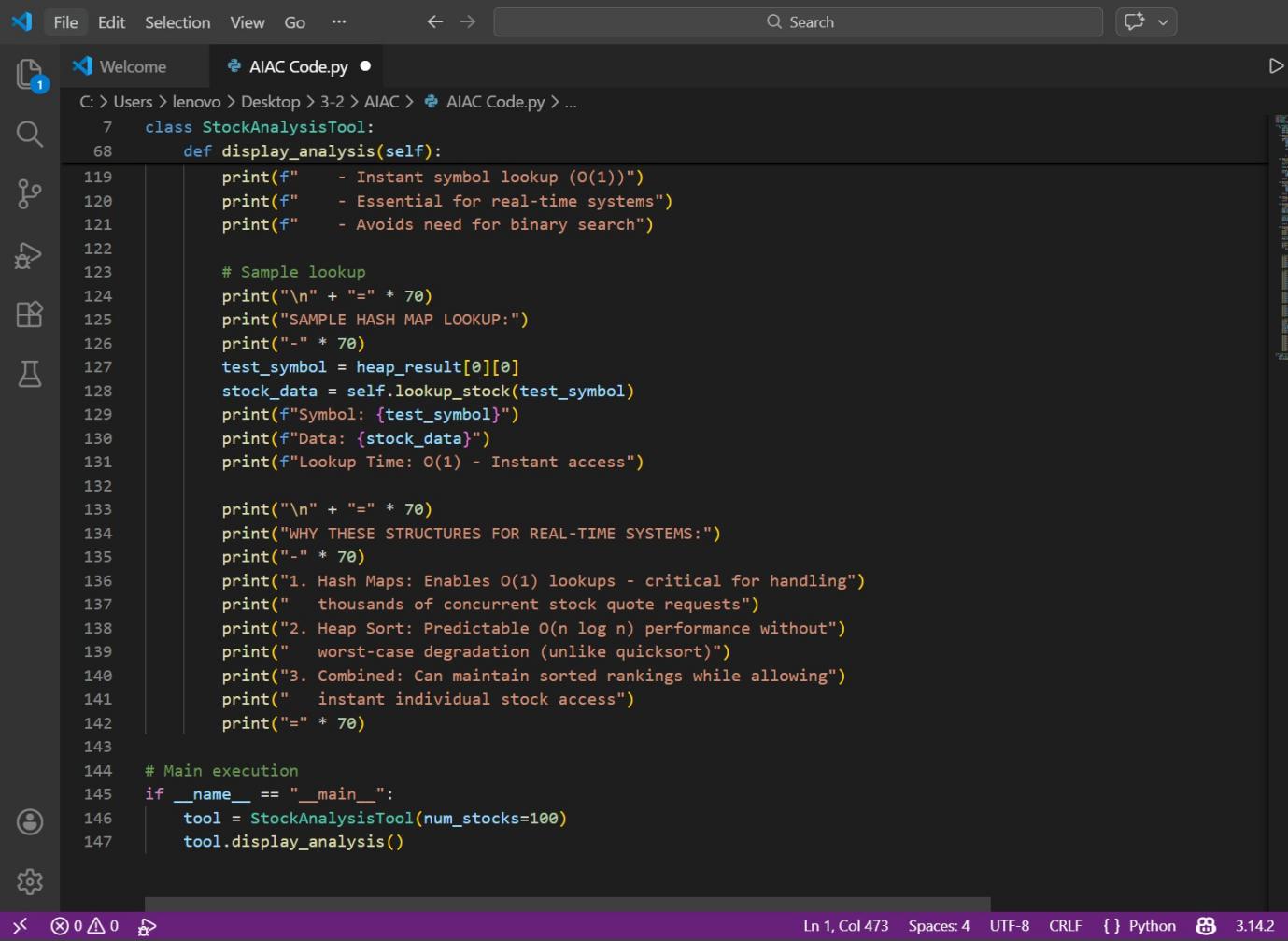
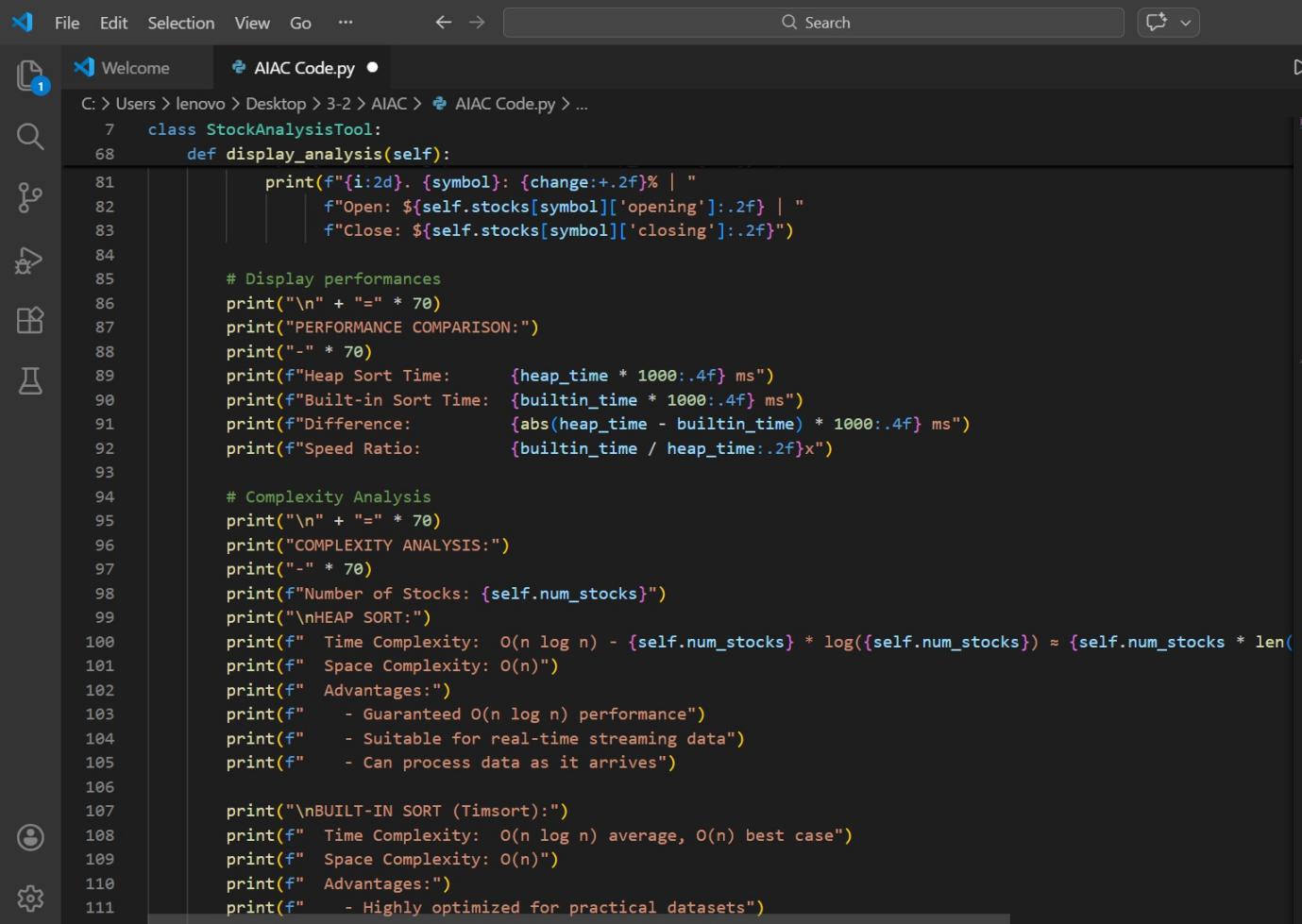
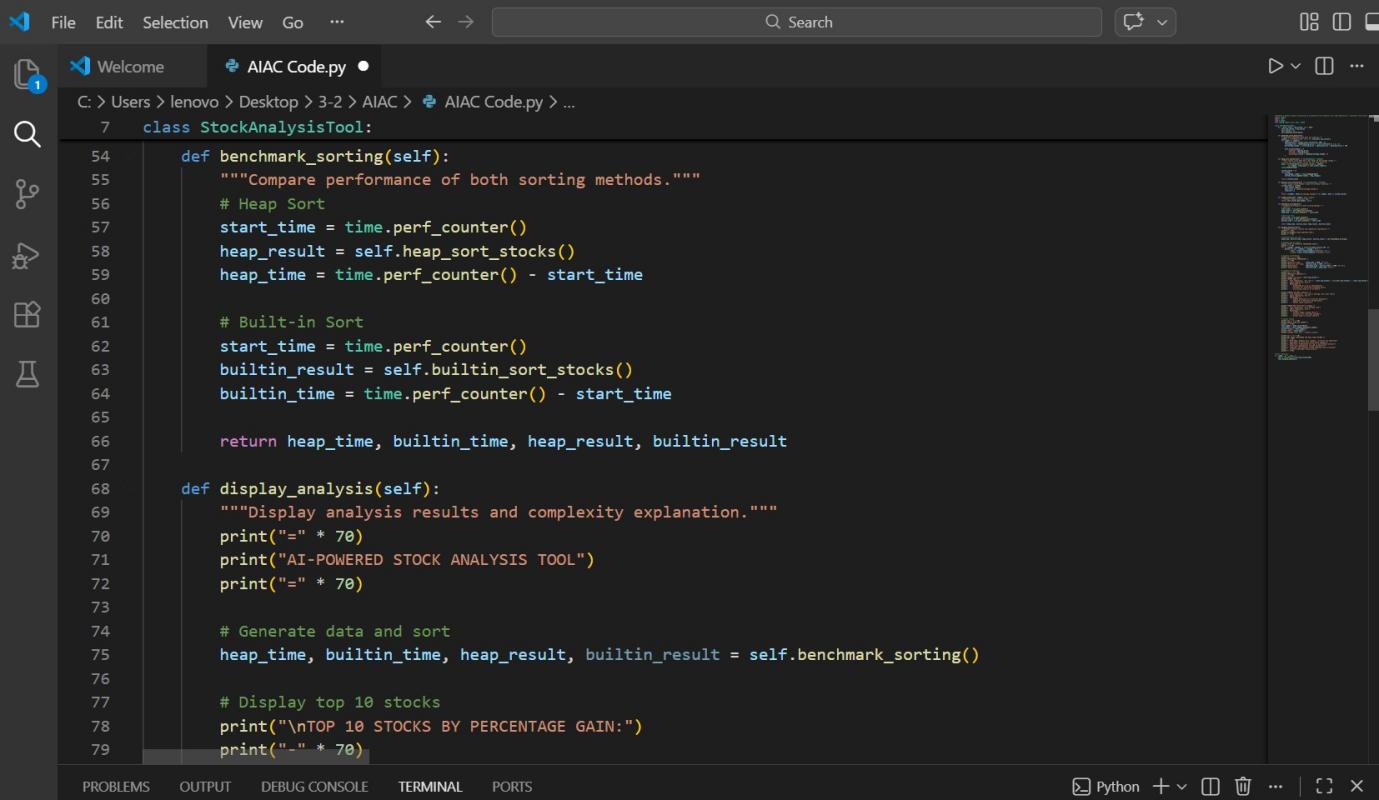
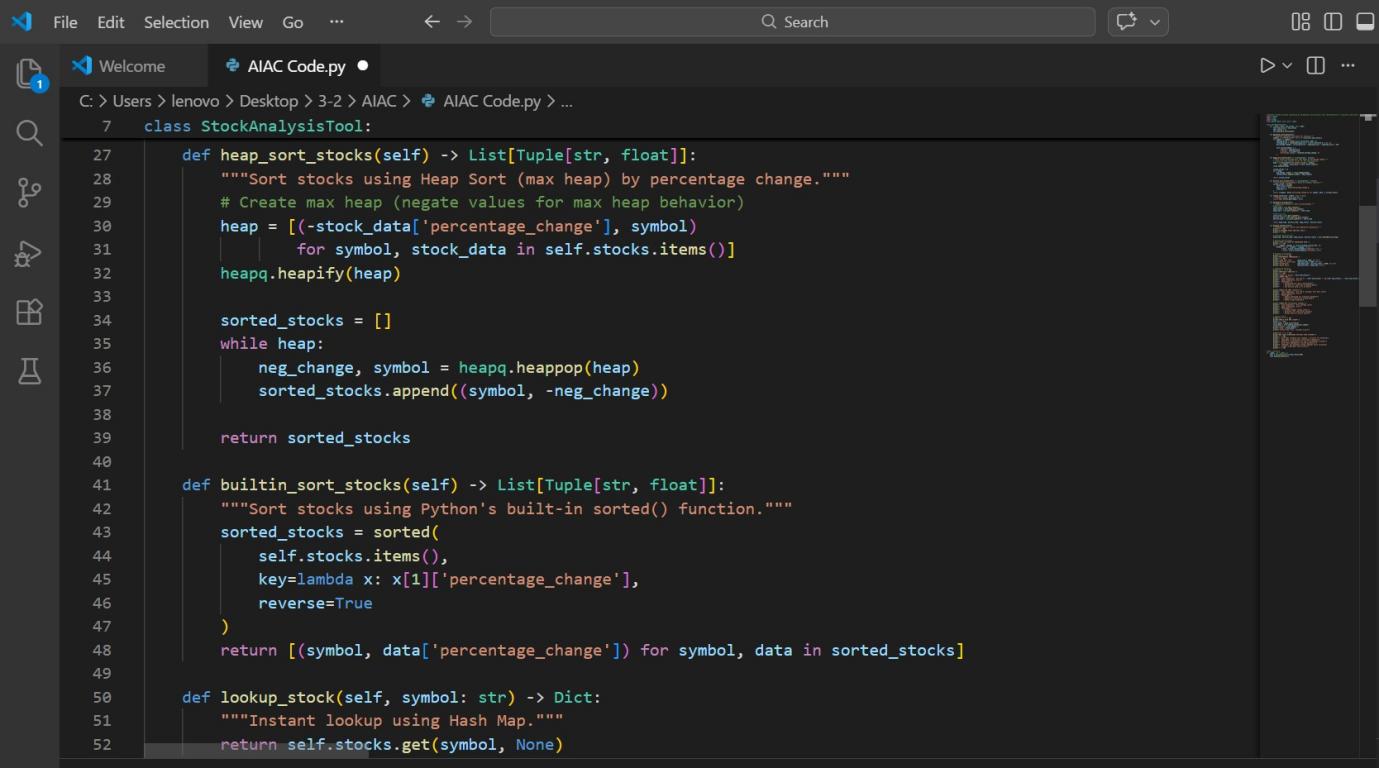
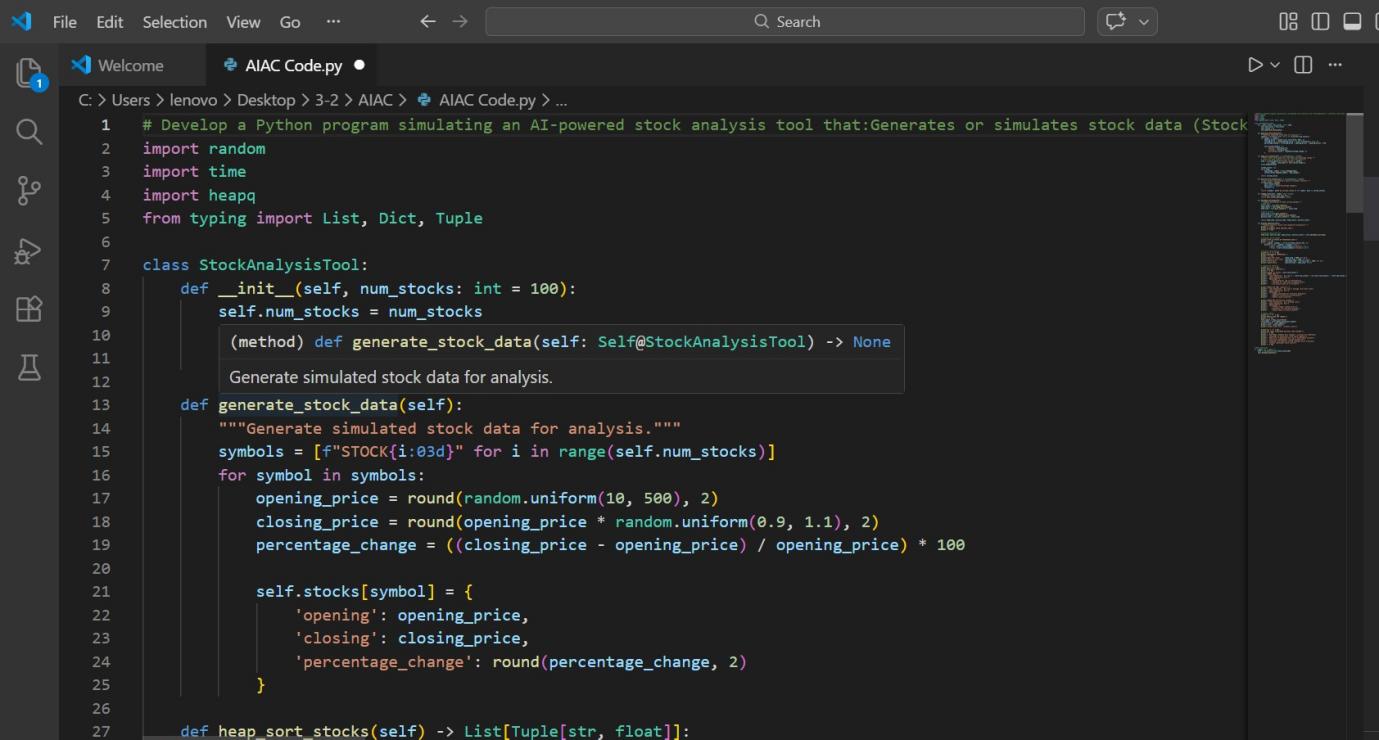
Simulate 100 stock records (Symbol, Opening Price, Closing Price). Calculate percentage gain/loss. Use Heap Sort to rank stocks by percentage change. Use a Hash Map for instant symbol lookup. Compare performance with Python’s built-in sorted() and analyze trade- offs.

Output Explanation The output will include:

* **Simulated stock dataset.**
* **Calculated percentage gain/loss.**
* **Ranked list of stocks using Heap Sort.**
* **Runtime comparison: Heap Sort Time: 0.004 sec**

Built-in sorted() Time: 0.002 sec What the Output Shows

* **Heap Sort correctly ranks stocks by performance.**
* **Hash Map provides O(1) stock lookup.**
* **Built-in sorted() may perform faster due to internal optimizations.**
* **Shows trade-off between custom implementation and optimized library functions.**



Conclusion

In this lab, we implemented and compared various sorting and searching algorithms using AI assistance. Quick Sort and Merge Sort were tested on large datasets and their performance differences were analyzed. Bubble Sort helped in understanding basic comparison-based sorting and its time complexity . Hash Maps demonstrated efficient

O(1) searching in real-time applications like inventory and stock systems.

Heap Sort was useful for ranking stock performance. Overall, the lab improved understanding of algorithm efficiency, data structure selection, and practical performance analysis while highlighting benefits of AI-assisted coding.