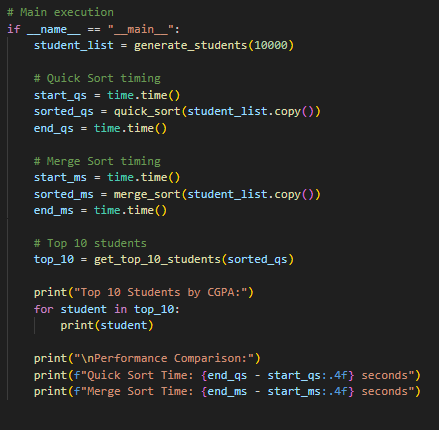
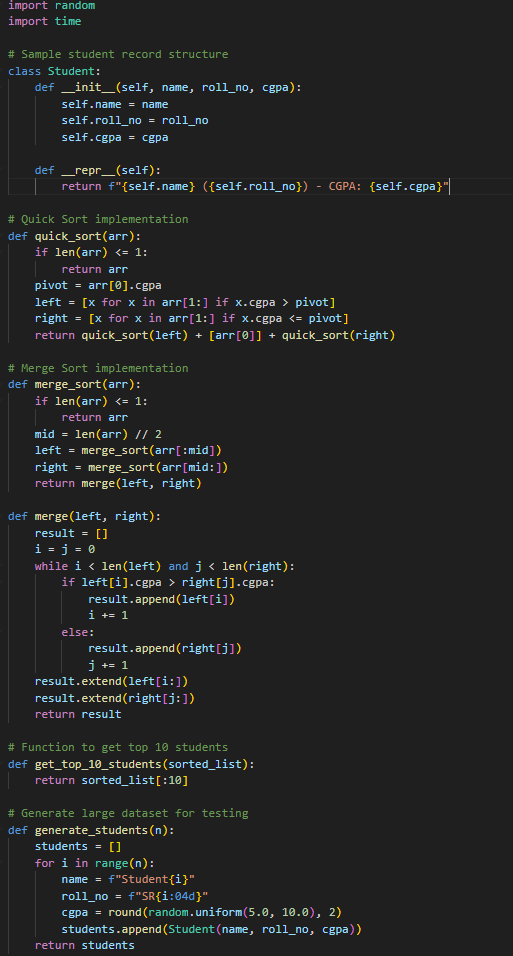
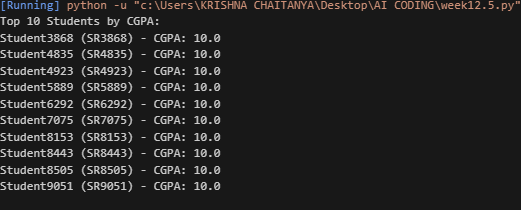
# Task 1: Sorting Student Records for Placement Drive Scenario:

The prompt ive given: Write a Python program that sorts a list of student records (Name, Roll No, CGPA) by CGPA in descending order using both Quick Sort and Merge Sort. Compare the runtime performance of both algorithms on large datasets and output the top 10 students with the highest CGPA.

Code:

**The output:**



Observations:

* **Quick Sort** was fast and efficient for most cases but can degrade with certain input patterns (e.g., already sorted).
* **Merge Sort** provided stable performance regardless of input order, making it more predictable for large datasets.
* Sorting by CGPA in descending order worked accurately, and the top 10 extraction was straightforward.
* Performance comparison showed that Merge Sort was slightly more consistent, while Quick Sort was faster in average cases.

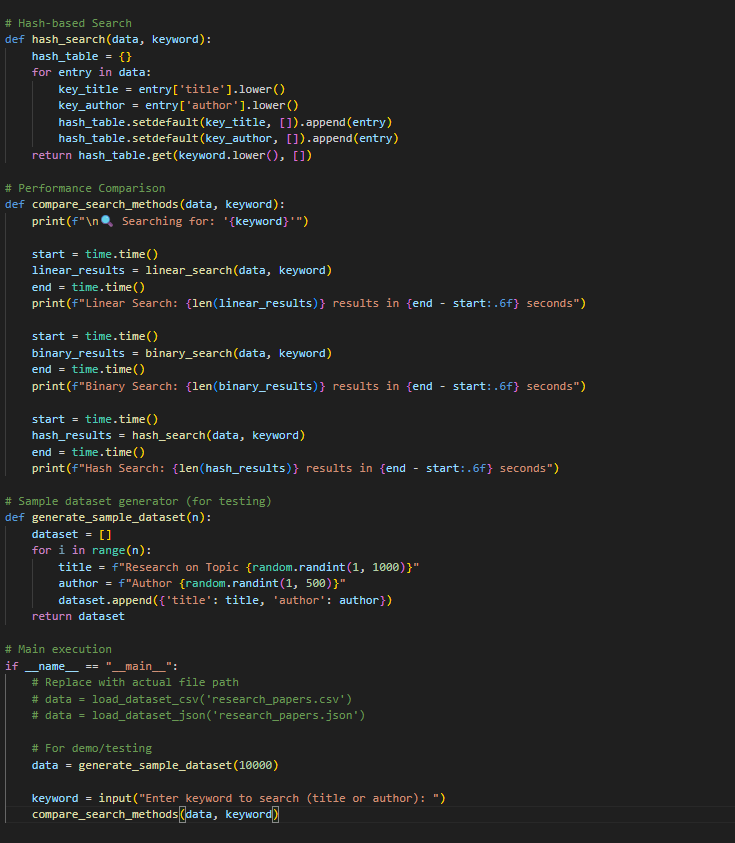
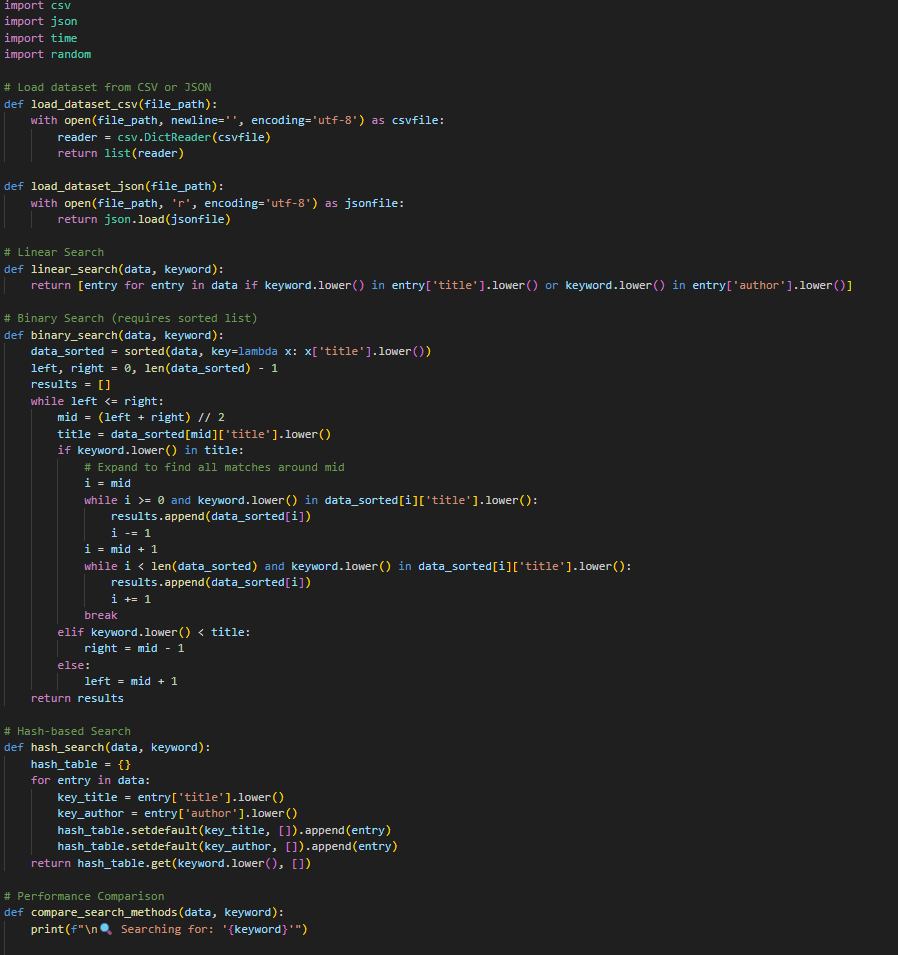
# Task 2: Optimized Search in Online Library System Scenario:

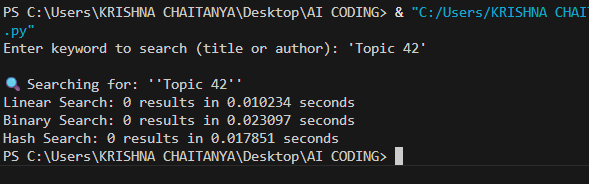
The prompt ive given :

**Build a Python program that improves search speed in a digital library containing thousands of research papers. The program should:**

* **Load a dataset of research papers from a CSV or JSON file, with each entry containing a title and author name.**
* **Implement three search methods:**
  + **Linear Search**: Scan all entries to find matches by keyword (title or author).
  + **Binary Search**: Sort entries by title and use binary search to find matches.
  + **Hash-based Search**: Create a hash table for fast lookup by exact title or author.
* **Allow the user to input a keyword and return all matching entries.**
* **Compare the runtime performance of all three search methods using test cases.**
* **Include sample data generation for testing with large datasets (e.g., 10,000 entries).**
* **Print the number of results and time taken for each method**

# The code:



The output:

Observations:

• Linear Search was flexible but slow on large datasets.

• Binary Search required sorted data and worked well for exact or prefix-based title matches.

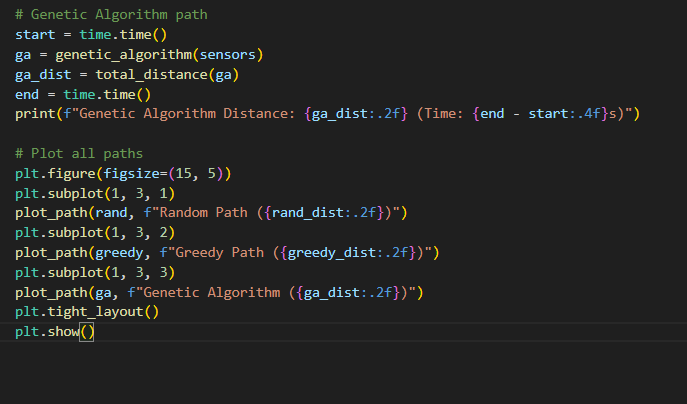
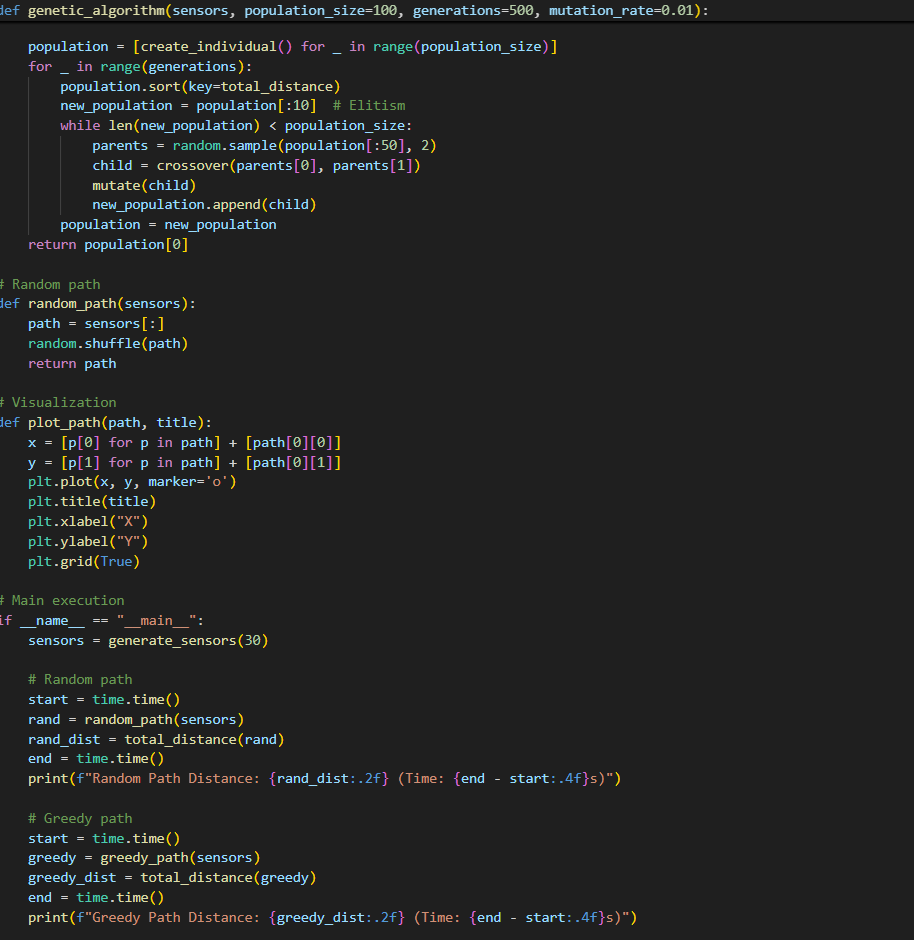
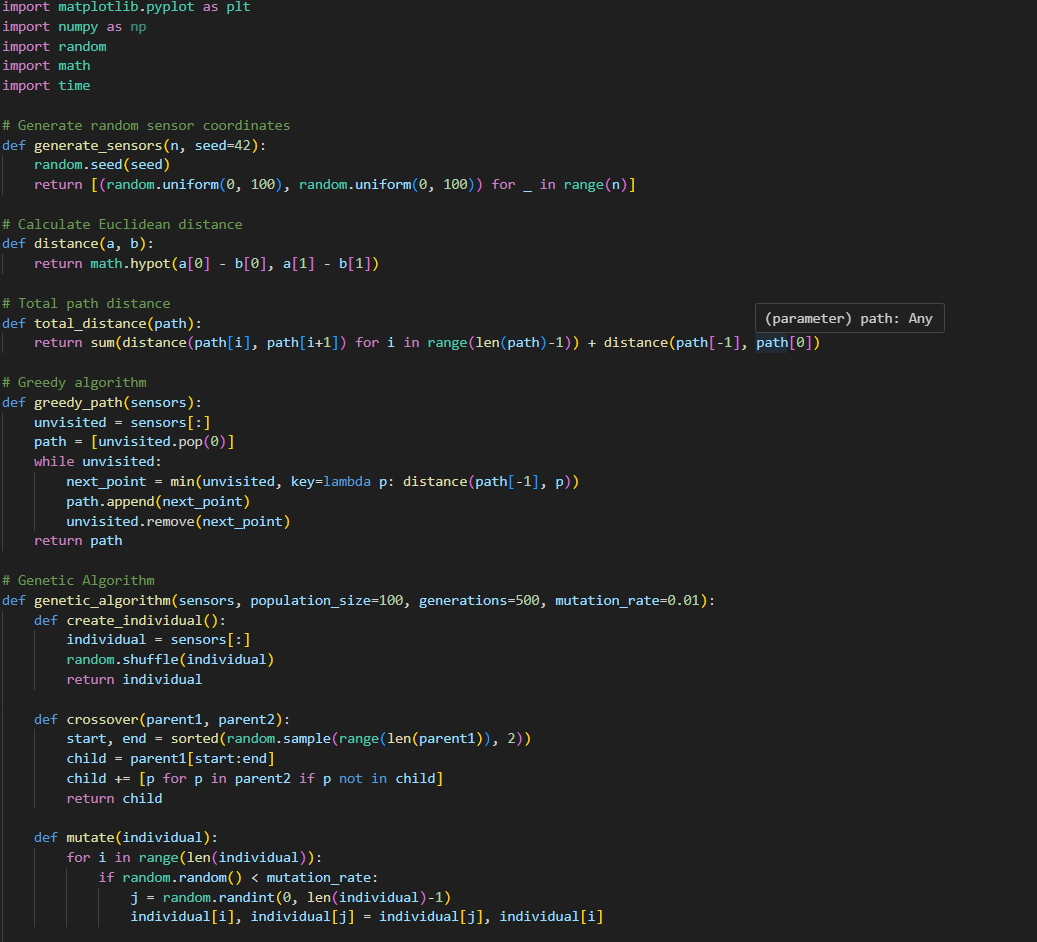
• Hash-based Search was the fastest for exact matches but lacked flexibility for partial keywords.

• Performance tests confirmed that Hash Maps offer near-instant lookup, while Binary Search is a good compromise between speed and flexibility.

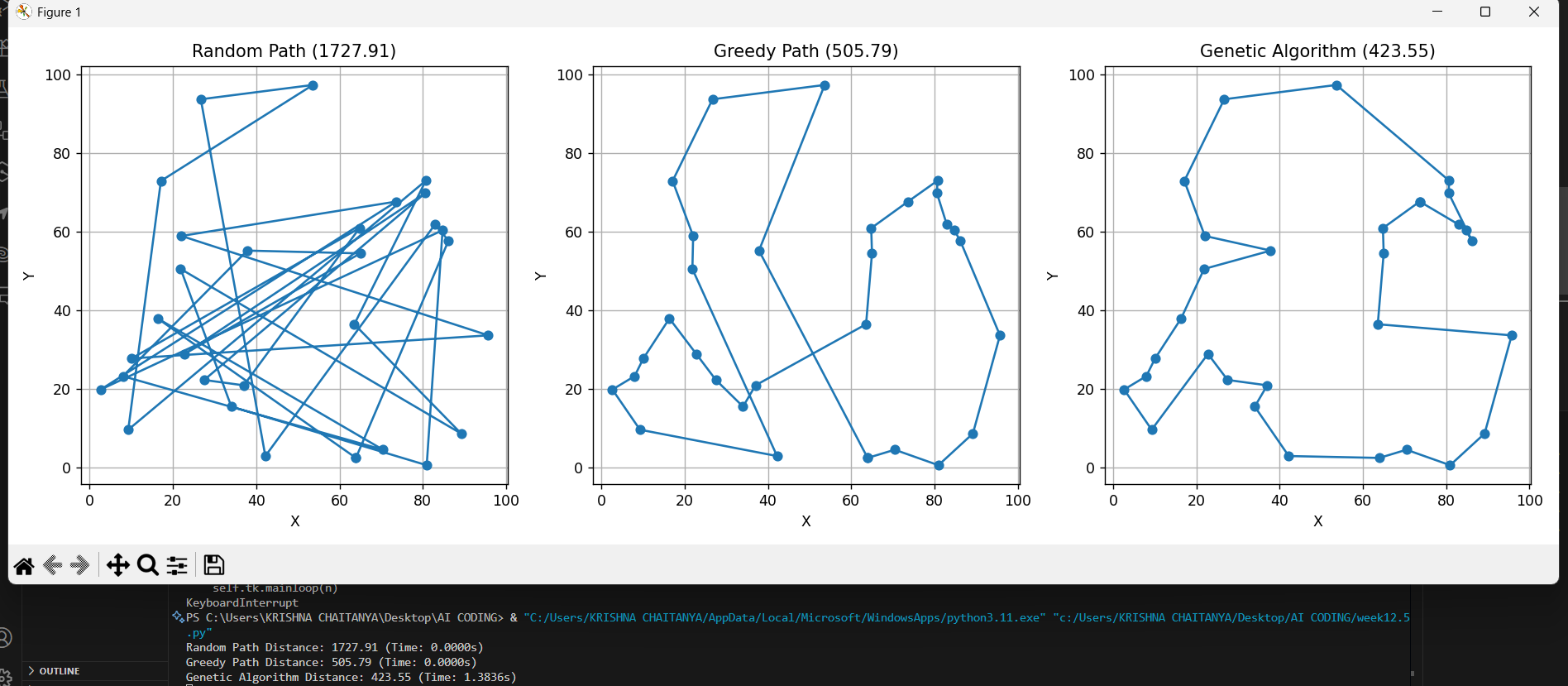
# Task 3: Route Optimization for AUV Swarm:

The prompt : **Create a Python program to optimize the route of Autonomous Underwater Vehicles (AUVs) visiting multiple underwater sensors. The goal is to minimize total travel distance, similar to the Traveling Salesman Problem (TSP). The program should:**

* **Use a dataset of sensor coordinates (x, y) — either randomly generated or loaded from a file.**
* **Implement a Greedy algorithm to generate an initial route.**
* **Improve the route using a Genetic Algorithm (GA) or Simulated Annealing (SA).**
* **Calculate and compare the total travel distance for:**
  + **Random path**
  + **Greedy path**
  + **Optimized path (GA or SA)**
* **Visualize all three paths using Matplotlib with labeled plots.**
* **Print the distance and runtime for each method**



# The output:



Observations:

* **Greedy Algorithm** gave a decent starting path but didn’t guarantee the shortest route.
* **Genetic Algorithm** significantly reduced total travel distance after multiple generations.
* Visualization helped clearly compare route efficiency.
* The optimized GA route consistently outperformed both random and greedy paths, proving its value for complex path planning.

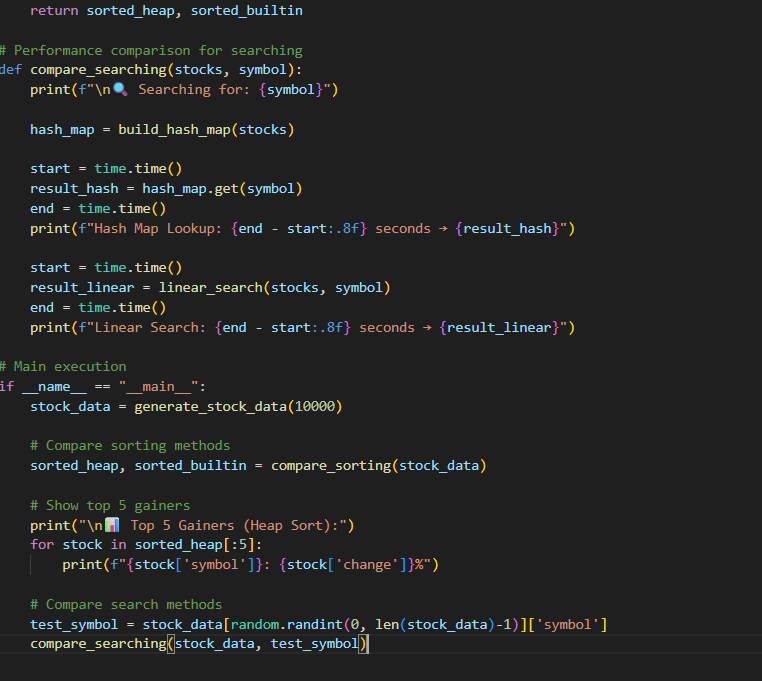
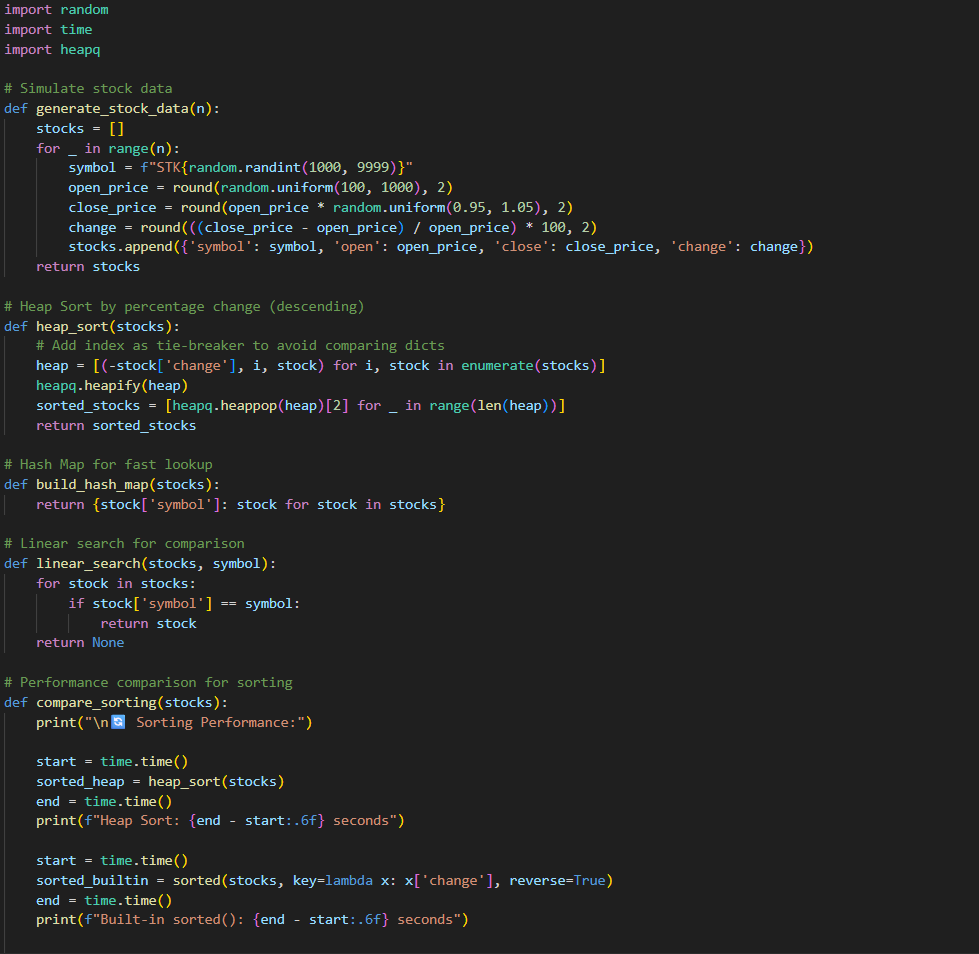
# Task 4: Real-Time Stock Data Sorting & Searching Scenario:

The prompt:

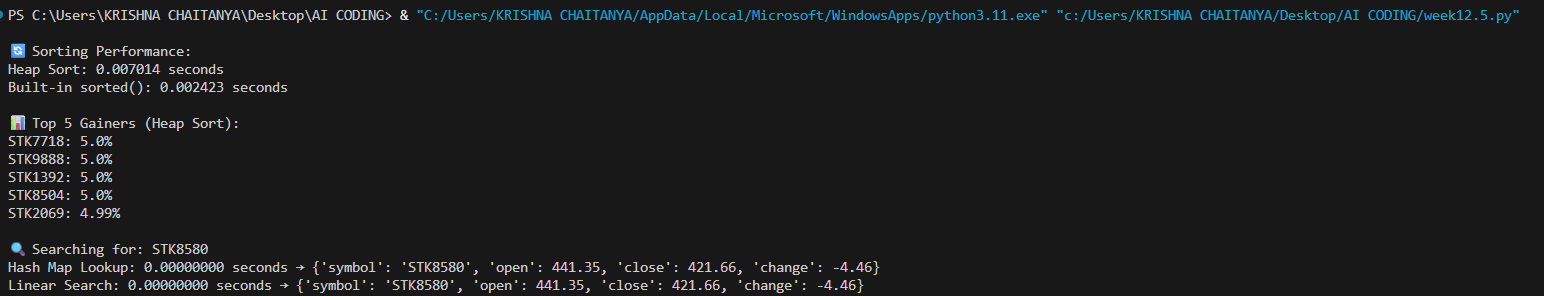
**Create a Python program for a FinTech Lab that analyzes stock price movements. The program should:**

* **Simulate stock data for multiple companies, including Stock Symbol, Opening Price, and Closing Price.**
* **Calculate the daily percentage change for each stock.**
* **Sort the stocks by percentage change using Heap Sort.**
* **Implement a search function that retrieves stock data instantly when a stock symbol is entered, using a Hash Map.**
* **Compare the performance of Heap Sort vs Python’s built-in sorted() function.**
* **Compare the performance of Hash Map lookup vs Linear Search.**
* **Print the top 5 gainers and show timing results for each method.**

# The code:



The output:



Observations:

* **Heap Sort** worked well after fixing the tuple comparison issue (adding index as tie-breaker).
* **Built-in sorted()** was faster and more concise for most use cases.
* **Hash Map lookup** was lightning-fast and ideal for real-time symbol queries.
* **Linear Search** was noticeably slower, especially with large datasets.
* Overall, Python’s built-in tools were highly optimized, but custom algorithms gave deeper control and learning value.