**Assignment 5: Quicksort Algorithm: Implementation, Analysis, and Randomization**

Student Name

University

Algorithms and Data Structures - Bi-term2

Professor Name

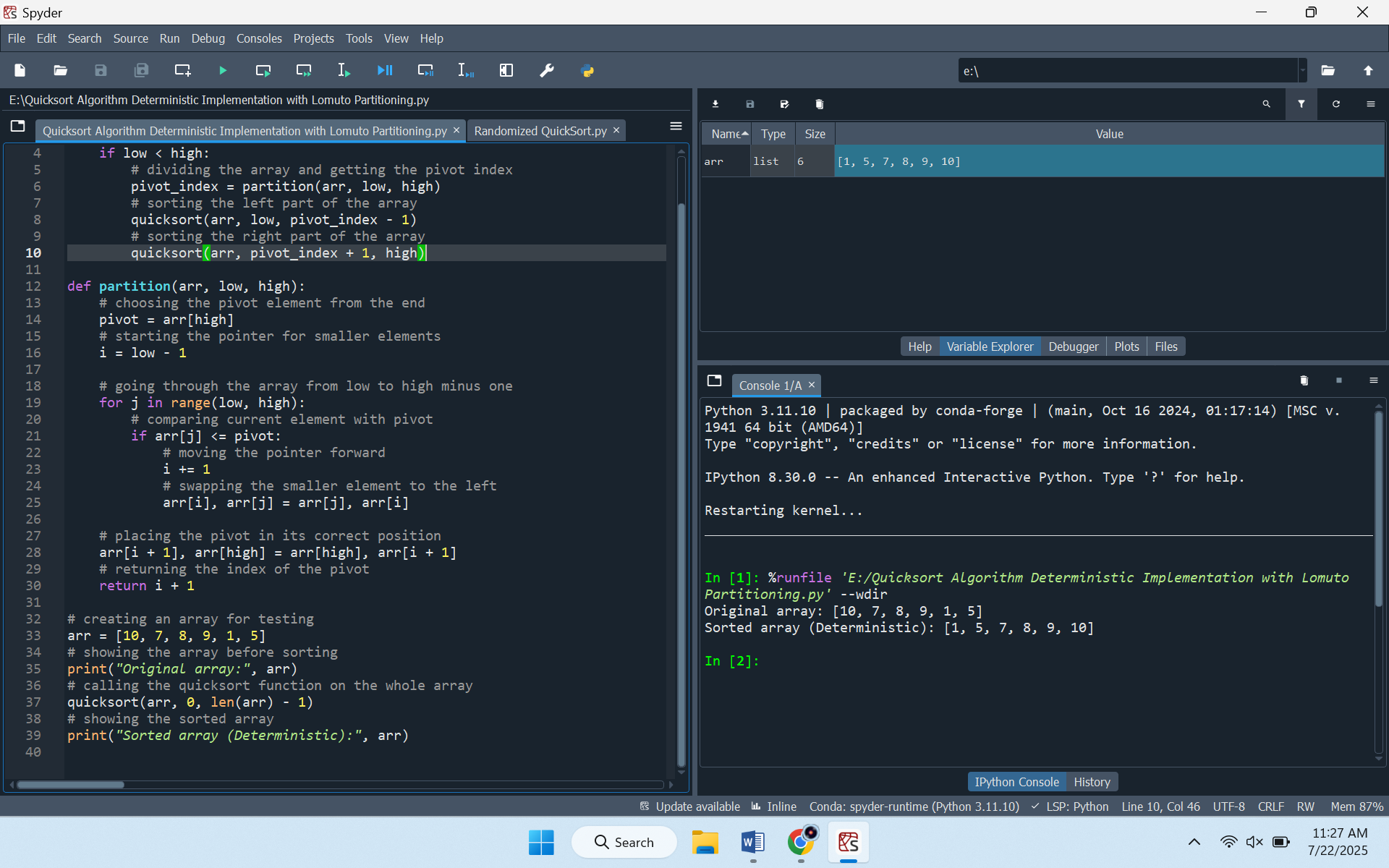
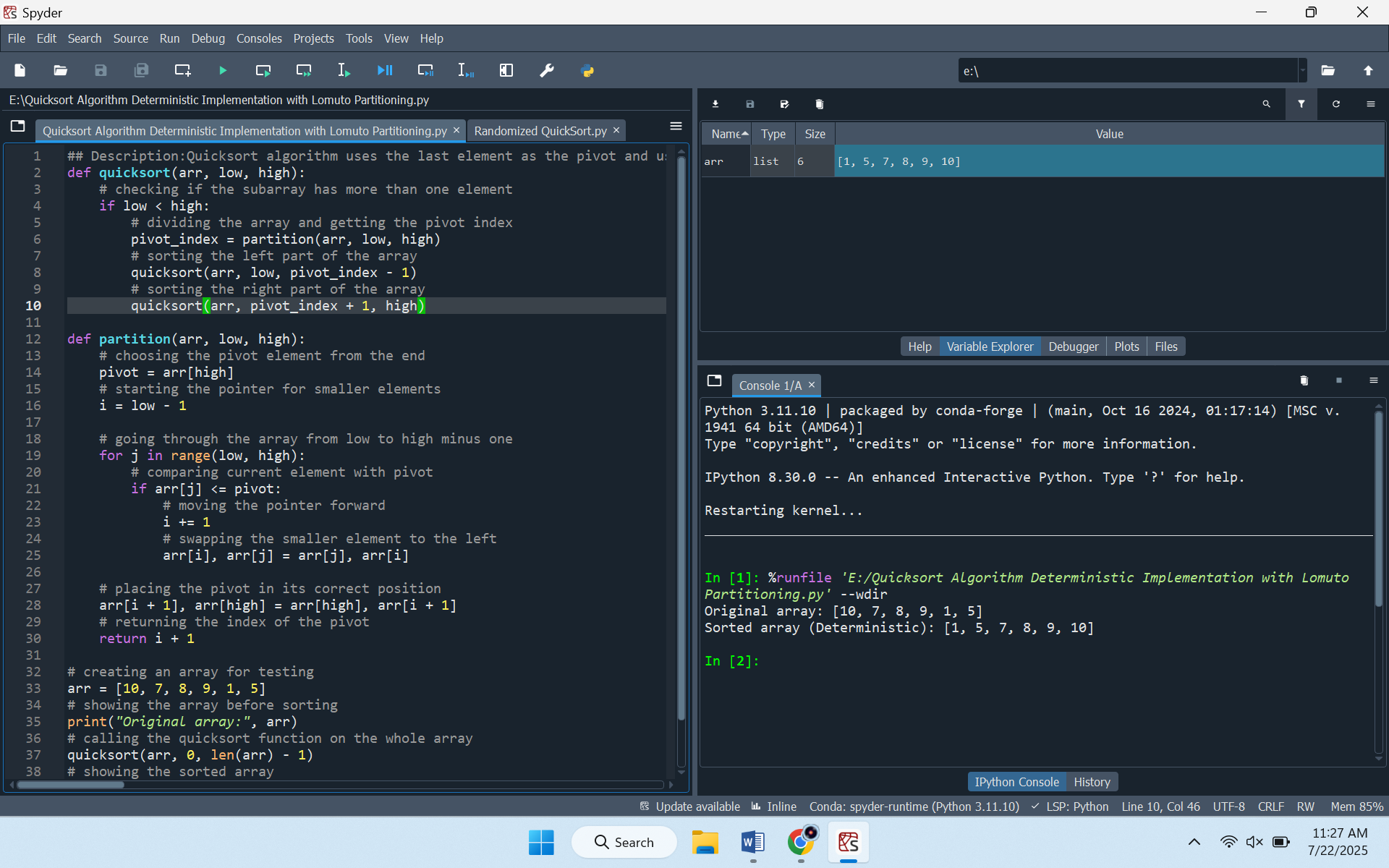
July 22, 2025

**Introduction**

Quicksort is an effective divide and conquer algorithm that is frequently used in current computing environments due to its advanced average-case performance. The algorithm involves sorting data by selecting a pivot element, after which it draws all other elements into order, followed by a recursive method of sorting the subarrays thus formed. A comparison of the deterministic method and the randomized method highlights the critical importance of the pivot selection strategy with regard to conclusively defining the overall success of the sorting method.

**Quick Sort Implementation**

Quicksort algorithm uses the Lomuto partitioning scheme and runs the last element as a pivot to restructure an array in place. In successive stages of recursion, elements are compared and interchanged until the subarray is correctly sorted. The process of partitioning ensures that all data elements that are less than the pivot are placed before the pivot, and the elements that are greater than this mark are placed in the next positions, forming smaller subarrays to be used in further descent.



**Time Complexity Analysis**

The best-case time complexity of quicksort is O (n logn), attained when each pivot partitions the input array into roughly equal halves. Its average-case complexity on unsorted input data is also the same, with the tendency to balanced partitions in practice. By contrast, the worst-case complexity is O (n2), realized when the partitions are highly skewed, an outcome frequently observed when the input is already sorted or in reverse order and the pivot choice is fixed.

**Reason for Average and Worst Case Complexity**

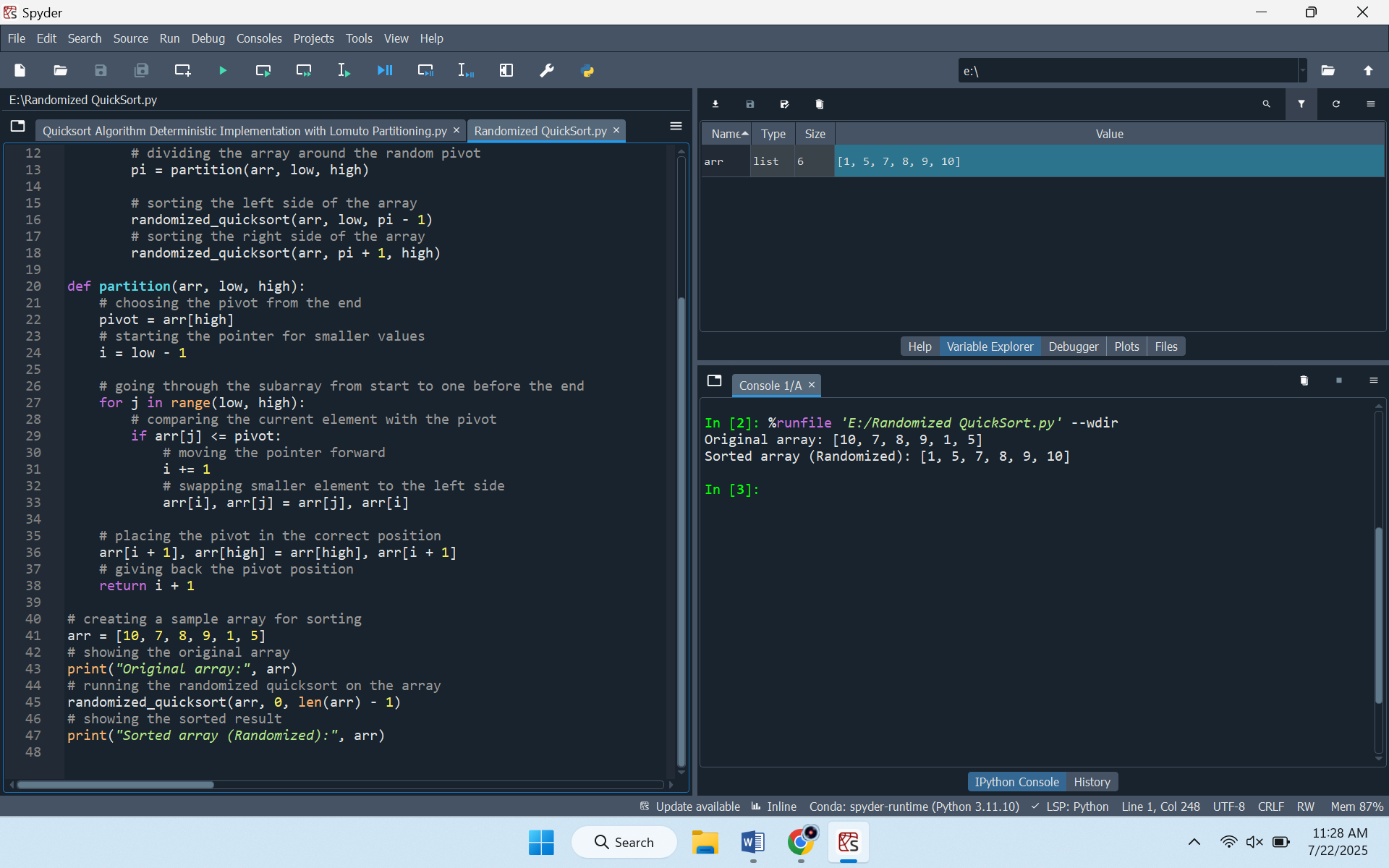
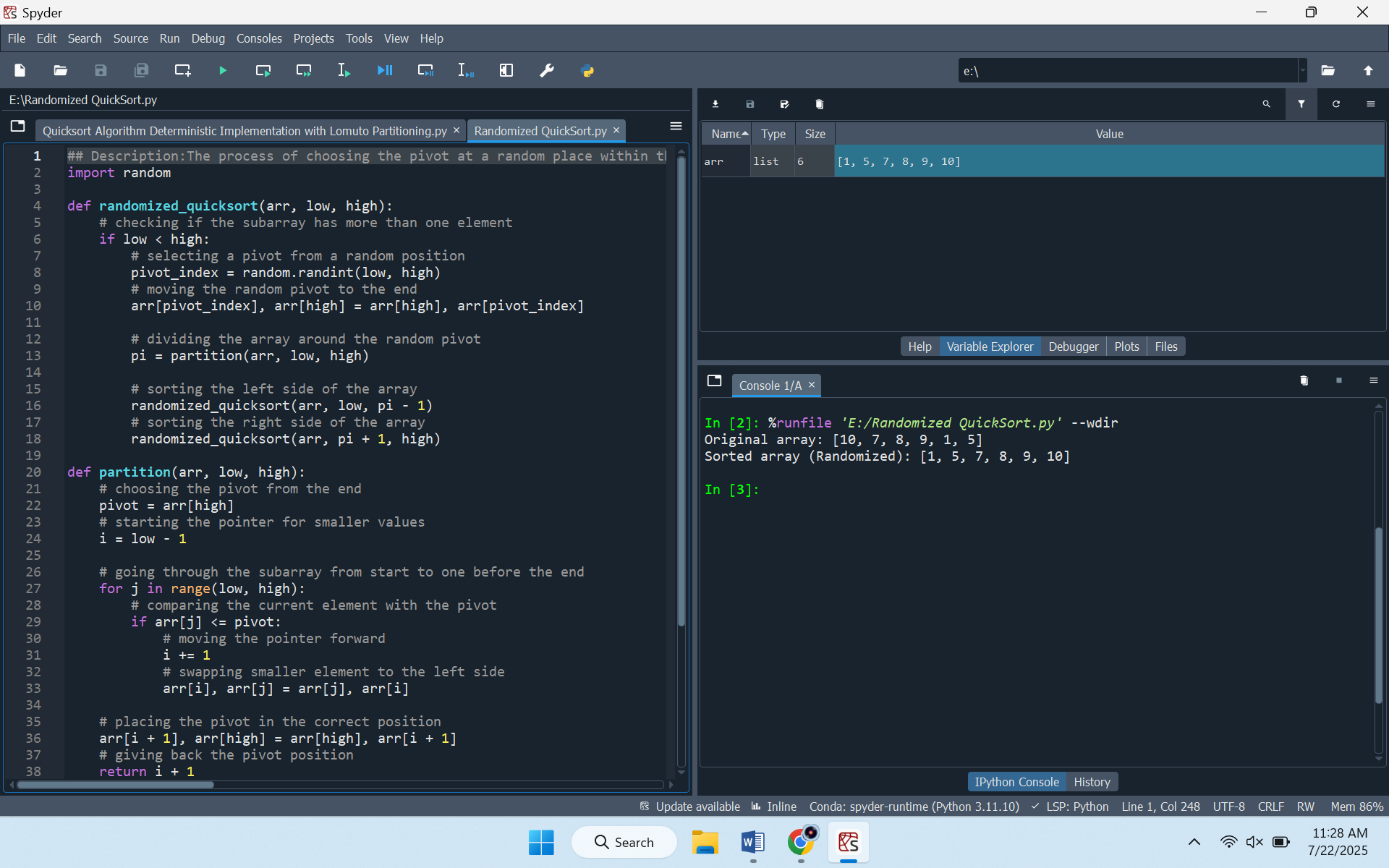
The average-case complexity of O (n log n) arises when the number of recursive levels is approximately log n, with each level containing n comparisons during partitioning. The worst-case complexity of O(n2) emerges when recursion depth approaches n. Here, every branch of the recursion leaves either side with essentially nothing, and as a result, the total amount of comparisons increases tremendously.

**Space Complexity and Overheads**

Space complexity is O(logn) for both average and best cases, as stack frames drive this bound in recursive traversals. In the worst case, the stack grows to O(n) due to deep recursion. However, this algorithm has only minor extra overheads due to managing the function calls, and no additional array is needed in the sorting, thus using resources efficiently.

**Randomized Quicksort Implementation**

Under the randomized forms of Quicksort, the pivot is chosen randomly in the subarray that may be reduced, which averts the pitfall towards continually unfavorable pivots that may cause an unbalanced partitioning.



**Performance Impact of Randomization**

Randomization also improves the practicality of Quicksort during its implementation, as it aids in maintaining an unpredictable approach to pivot selection. This prevents the same thing of choosing the worst-case pivot recursively, a habit that tends to appear when the input data is sorted or organized. Consequently, the algorithm resists consistently unbalanced partitions and thereby sustains the expected time complexity of O (n logn) with high probability.

By disrupting deterministic patterns in input, randomization fortifies the algorithm’s resilience to input order, markedly diminishing the probability of provoking the worst-case complexity of O (n2).

**Empirical Comparison of Quick Sort Variants**

Running times of deterministic and randomized Quicksort are compared under random, sorted, and reverse-sorted input datasets. Unit testing starting with arrays with various degrees of increment is done, to measure the performance scaling relative to data size and organization. Execution times are captured in the same condition of hardware to encourage uniformity.

The results confirm that the two algorithms exhibit comparable efficiency on random input, as the complexity approximates O (n logn) for each. On sorted or reverse-sorted data, deterministic Quicksort is associated with frequent but inevitable cases of poor pivot choice, resulting in considerably slower running times. By way of comparison, randomized Quicksort does not exhibit worse execution times when given any particular input, since worst-case partitions are less likely to occur. These observations confirm theory expectations, and these results demonstrate the benefits of random selection of pivot as a means of accomplishing better average performance as well as a reduction in chances of suffering tragic worst cases.

**Conclusion**

The actual implementation and overall analysis of Quicksort provided useful information based on both theory and practical implications of an efficient algorithm. The use of randomized pivot selection proved to have obvious performance benefits in its consistency of performance and reduced exposure to worst-case complexity over a wide range of input distributions. These results are an important confirmation of the utility of balanced partitioning in achieving the highest sorting efficiency.