**Week 3 Assignment 3: Understanding Algorithm Efficiency and Scalability**

Prasanthi Pakalapati

University

Algorithms and Data Structures - Bi-term2

Professor Name

July 16, 2025

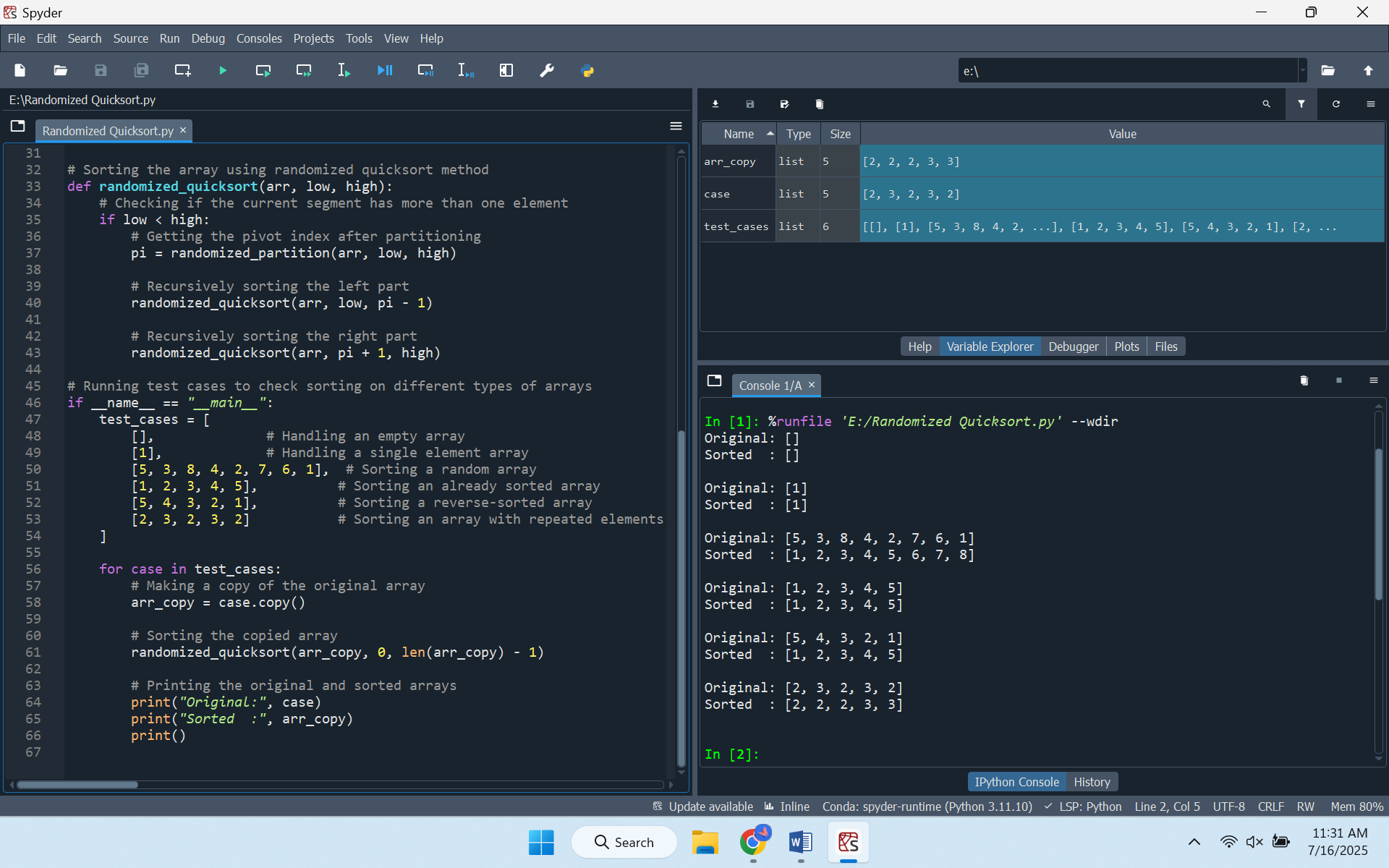
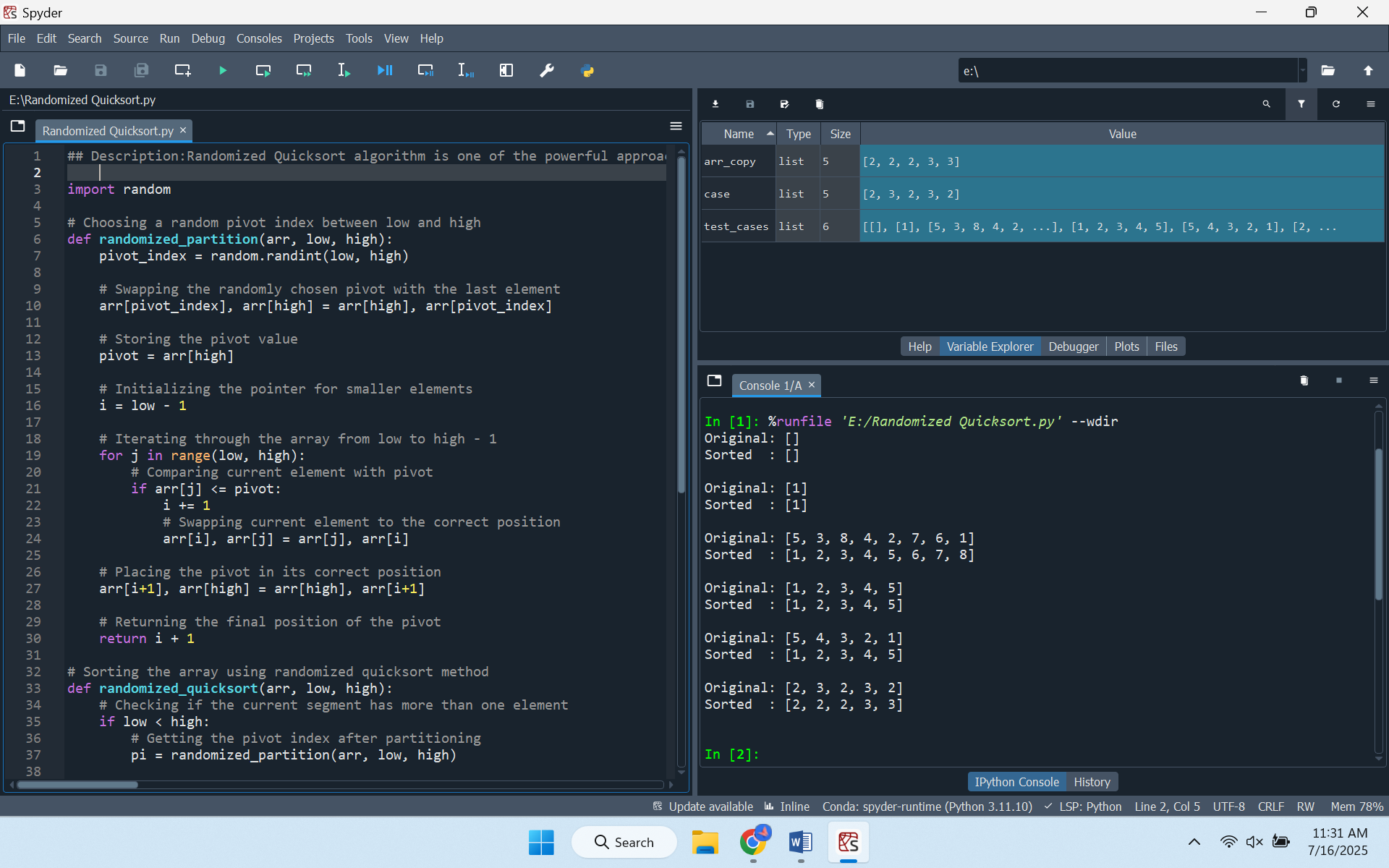
**Randomized Quicksort**

The Randomized Quicksort algorithm is a promising refinement of the well-known Quicksort technique in that it includes a random choice selection of the pivot element, thus decreasing the likelihood that the partitioning process produces biased sub-arrays. The randomized choice of pivot results in a more even distribution of data, hence a better performance compared to the average-case scenario. It is then recursively split into two sub-arrays of elements less than and greater than the pivot, respectively, and the two sub-arrays are sorted recursively. Randomized Quicksort, due to its divide-and-conquer style and its average-case performance, is specifically useful with large sets and is commonly used in a system where quick sorting is a necessity.

**Implementation of Randomized Quicksort**

Randomized Quicksort is a probabilistic version of Quicksort whose correctness is based upon the uniform random choice of a pivot element within the current partitioned subarray. The algorithm is compensated by the probabilistic complexity that might occur when the wrong pivot is used by introducing randomness and reducing the likelihood of the worst-case behavior of the algorithm. Therefore, its real-world use needs to be made as robust as possible to face the following difficult edge cases: arrays containing repeat elements, empty arrays, and arrays already ordered either in ascending or descending order. Efficient treatment of these situations forms the basis of the maintenance of consistent performance with different input conditions.

**Implementation**

Randomized Quicksort is a very effective sorting algorithm in which a pivot element is chosen randomly and divides the original array, and subsequent subarrays are then sorted using the same algorithm in a Recursive fashion. Relative to traditional Quicksort, this variation reduces the chances of experiencing the worst-case performance. Because element rearrangement is directed by a randomly chosen pivot, the algorithm consistently achieves an average-case time complexity of O(n logn), thereby delivering dependable performance across a broad spectrum of input conditions, ranging from empty arrays and single-element arrays to already sorted sequences, reverse-ordered sequences, and datasets containing repeated values.

**Time Complexity Analysis**

The average-case time complexity of Randomized Quicksort is O (n log n). This finding is proven by methods based on recurrence and probabilistic methods, using in particular indicator random variables. The expected cost of comparisons incurred in sorting is proportional to n log n and is due to the random choice of pivots, which, on average, balances the partitions. In contrast to the deterministic variant, where suboptimal pivot choices can deteriorate performance to O(n2), the randomized approach statistically avoids unbalanced partitions, thereby producing more consistent and efficient execution.

**Empirical Comparison and Performance Evaluation**

The empirical analysis of theoretical expectations should involve a comparison between the Randomized Quicksort algorithm and the deterministic version of this algorithm, where the first element is always selected as the pivot. This should be done by comparing various distributions of input examples, randomly generated arrays, sorted arrays, reverse-sorted arrays, and repetition arrays. It is expected that, with such conditions, Randomized Quicksort would demonstrate a unanimously higher level of performance than its deterministic variant, with the former mostly showing a stronger performance contrast when deterministic pivot selection leads to the formation of unbalanced partitions (Khursheed et al., 2024). It must also investigate the variance between empirical results and theoretical expectations, reflecting on the system-level overhead, the spread of inputs, and implementation specifics.

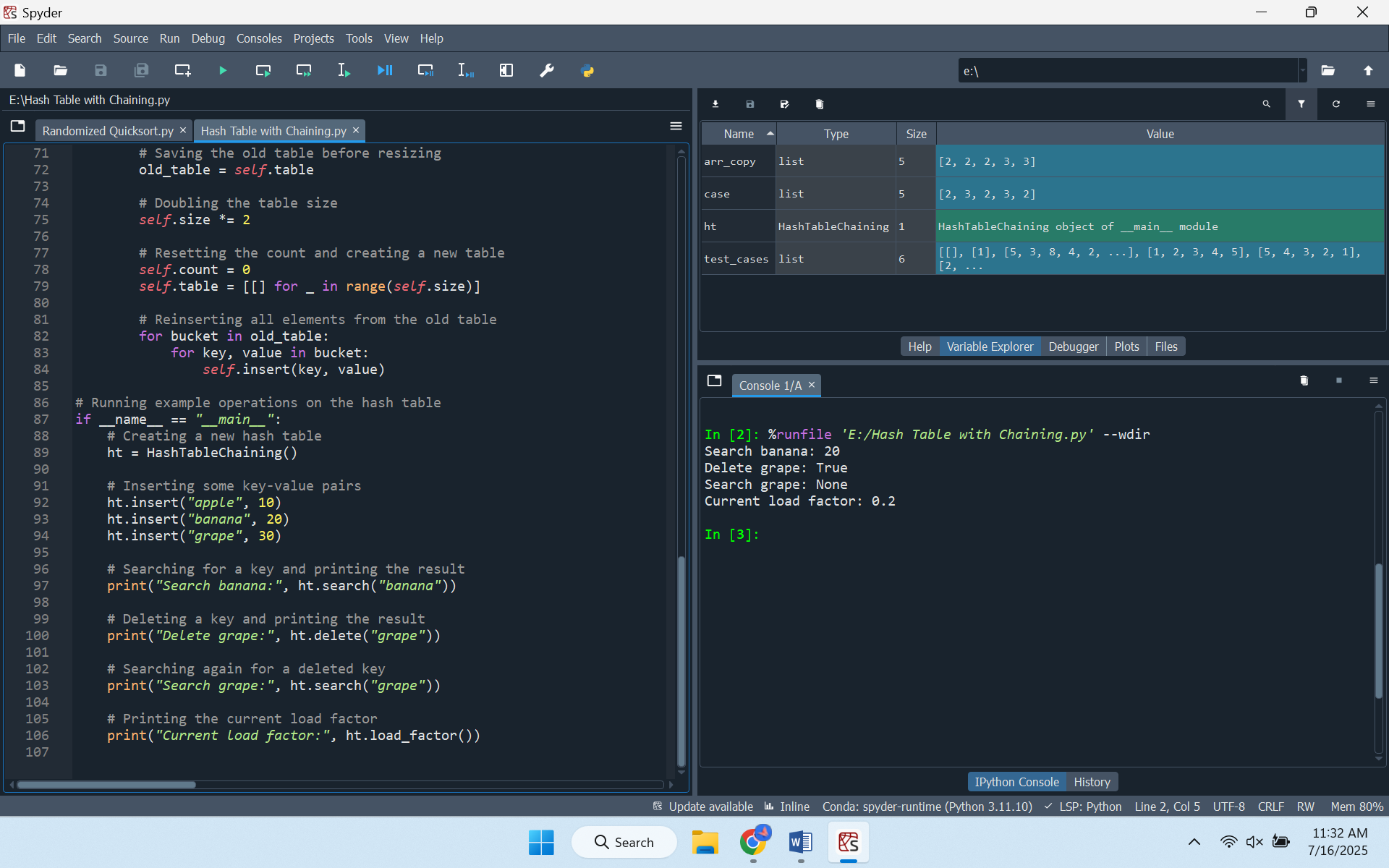
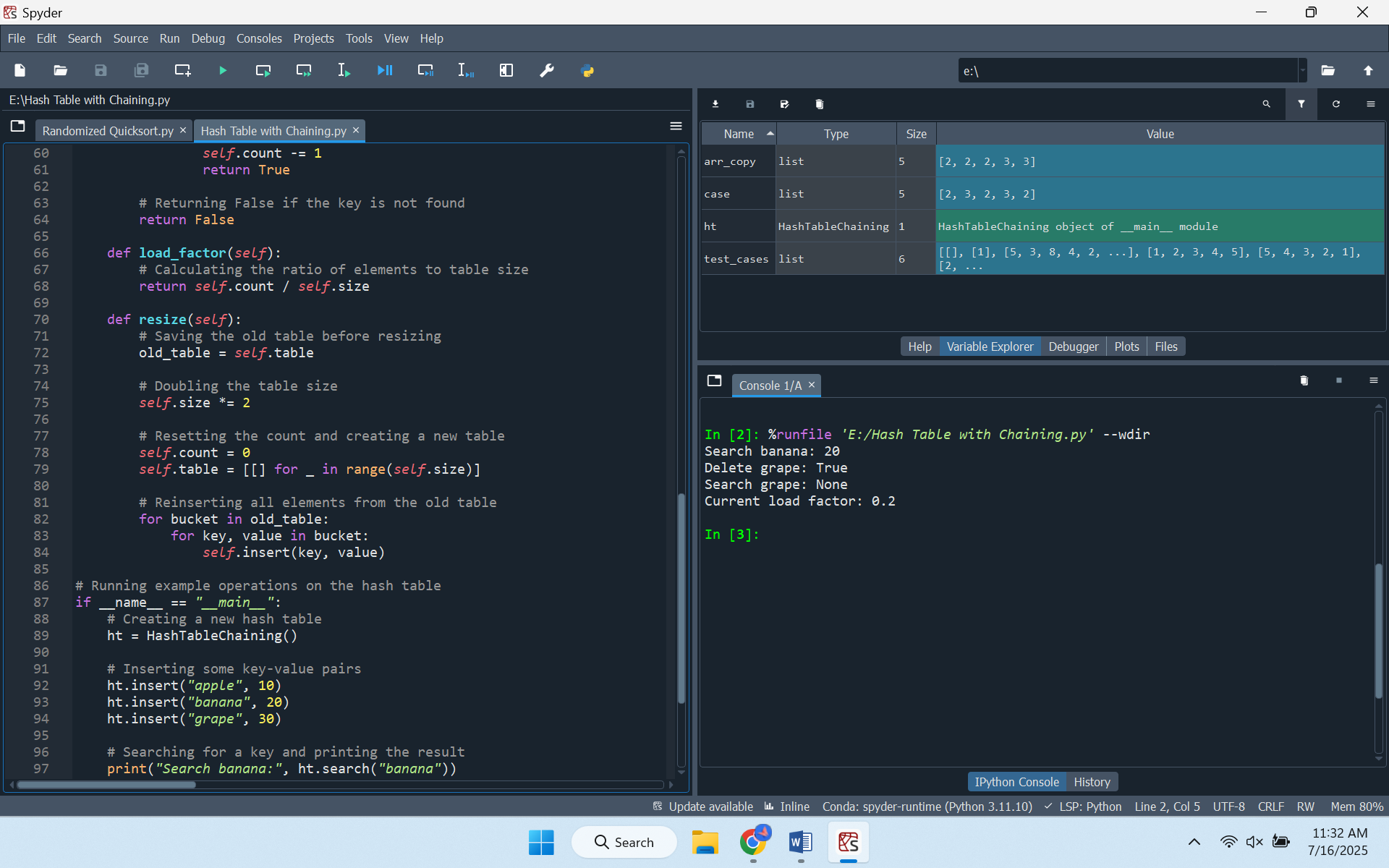
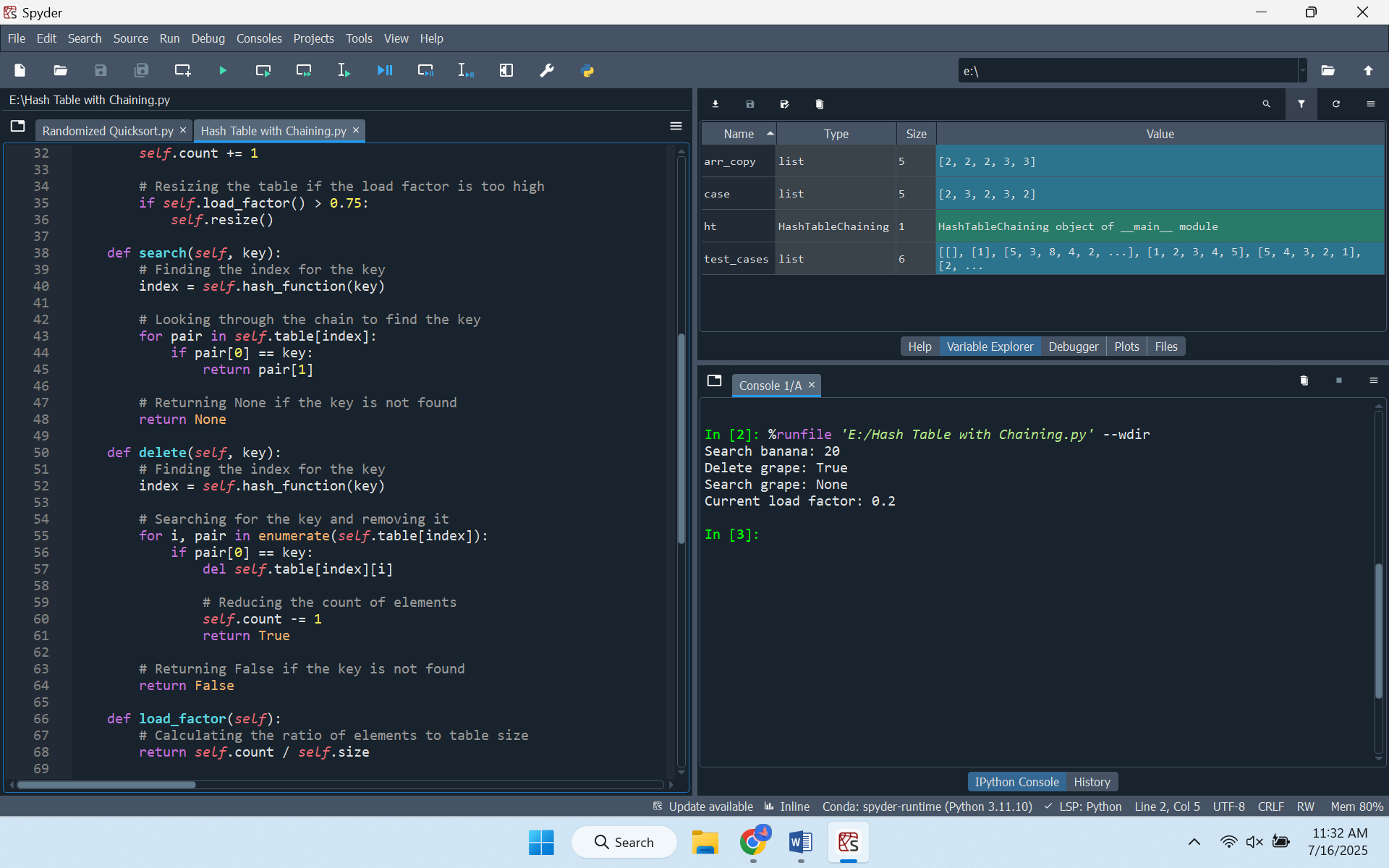
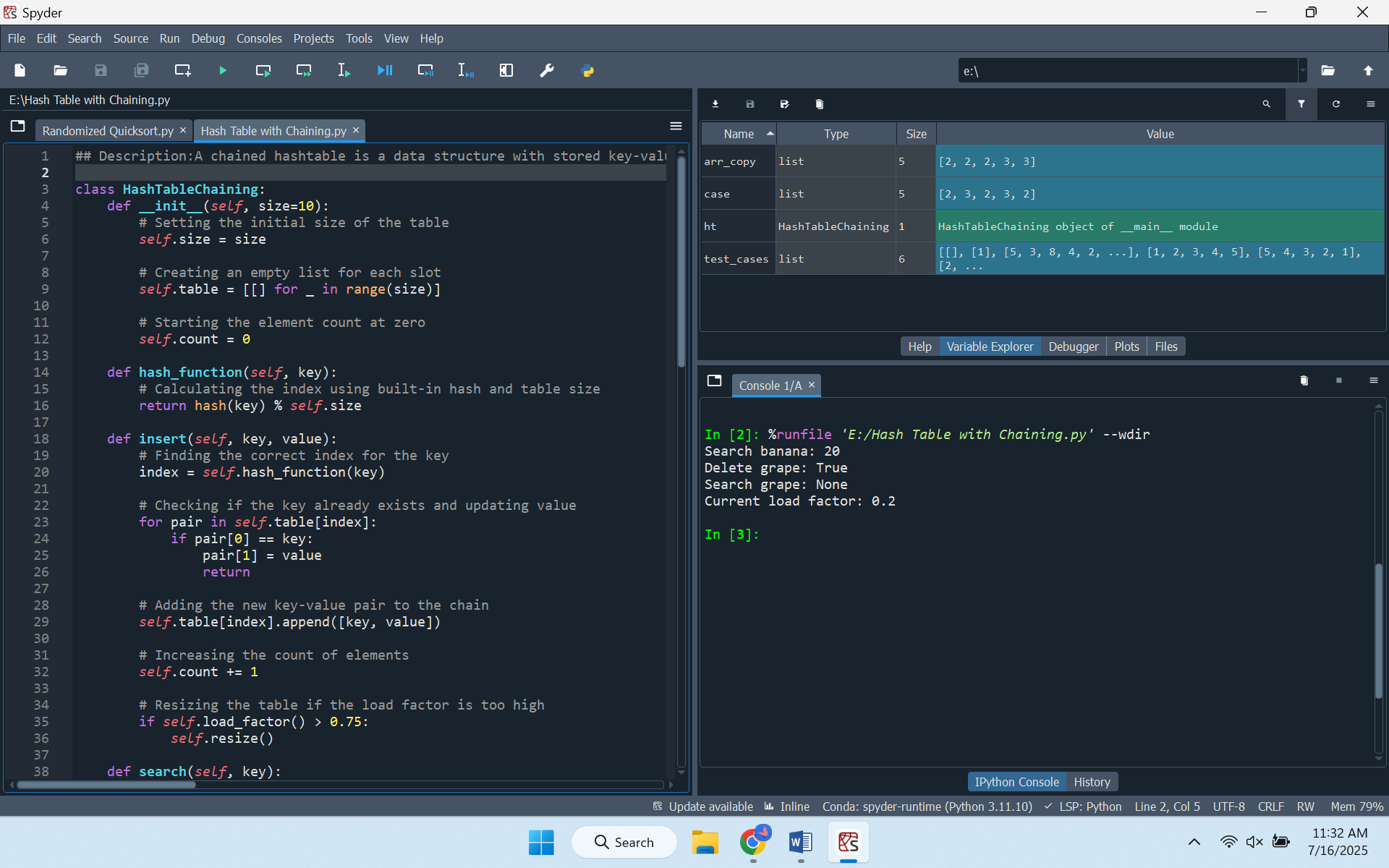
**Hashing with Chaining**

Hashing with Chaining is a collision-handling mechanism used on hash tables, designed to handle cases when several keys are assigned to the same index. The approach uses a linked list, or other similar data structure, to collect together all items that hash to the same point, instead of just placing the individual value in a specific place. Whenever a new item is inserted and the resulting collision is identified, it is added to the respective disk at the correct location. This structure maintains efficient average-case performance in insertion, deletion, and search requests, especially where the load factor is maintained low. Hashing with Chaining allows this by allowing various items to share an index, providing a dynamic and useful means of handling collisions in dynamic storage solutions.

**Implementation of Hashing with Chaining**

Traditional hash-table systems that use chaining are vital in eliminating collisions. With this approach, several key-value pairs can be stored within one slot and are stored as a chain; this way makes it easier to handle collisions other than overwriting entries. To gain uniform key distribution and, consequently, to reduce the risk of collisions, one should select a strong hash function that is preferably a universal family (Russinovich & Salem, 2024). It is also necessary that the structure could effectively manage basic operations: insertion of new key-value pairs to the correct chain, search that finds values that are linked with specified keys, deletion that removes chain elements, but does not necessarily diminish the structure.

**Implementation**

A chained hash table is an effective data structure in that resolving collisions is achieved by ensuring linked lists of key-value pairs holding each index. It calculates an index on every key and uses a hash function on that key, and in case of collision, it will store more than one pair at a time. The table does this updating during insertion, by checking whether the desired key is already there or not, and updating its value where it is, or adding a new pair where it is not. To search involves finding the proper index, then scanning the chain to extract values. Deletion identifies a certain key-value and deletes it within the chain. The load factor, as the division between the number of stored elements and table size, is monitored continuously, and automatic resizing is launched when it exceeds the preset limit is launched. When the triggers are resized, the table size is extended twice, and everything is inserted once again to distribute these equally and hence the efficiency in terms of operations.

**Performance and Efficiency Analysis**

Within the framework provided by simple uniform hashing, the average time complexities of search, insert, and delete operations are constant, denoted by O (1 + α), where α represents the load factor. Load factor is the ratio between the number of stored elements and the total number of slots and could directly influence the average chain length, and therefore affect the operational performance as well. By increasing the load factor, there will be long chains, and consequently, the average time taken to accomplish operations will be increased. Therefore, to guarantee the best results, there must be a means of sustaining a low load factor, like dynamically resizing the hash table when a certain limit is exceeded. Such resizing entails rehashing the already existing elements into a bigger table, hence reorganizing the keys and shortening the chain length to promote its effectiveness.

**References**

Khursheed, M. S., Khan, M. J., Haider, R., & Enayet, A. (2024, October). Threshold Optimization and Hashing vs Radix vs QuickSort] Threshold Optimization and Comparative Performance Evaluation of Quicksort Algorithms: Hashing vs Radix vs QuickSort. In Proceedings of the 2024 8th International Conference on Algorithms, Computing and Systems (pp. 124-130).

Russinovich, M., & Salem, A. (2024). Hey, That's My Model! Introducing Chain & Hash, An LLM Fingerprinting Technique. arXiv preprint arXiv:2407.10887.