**Assignment 4: Heap Data Structures: Implementation, Analysis, and Applications**

Prasanthi Pakalapati

University

Algorithms and Data Structures - Bi-term2

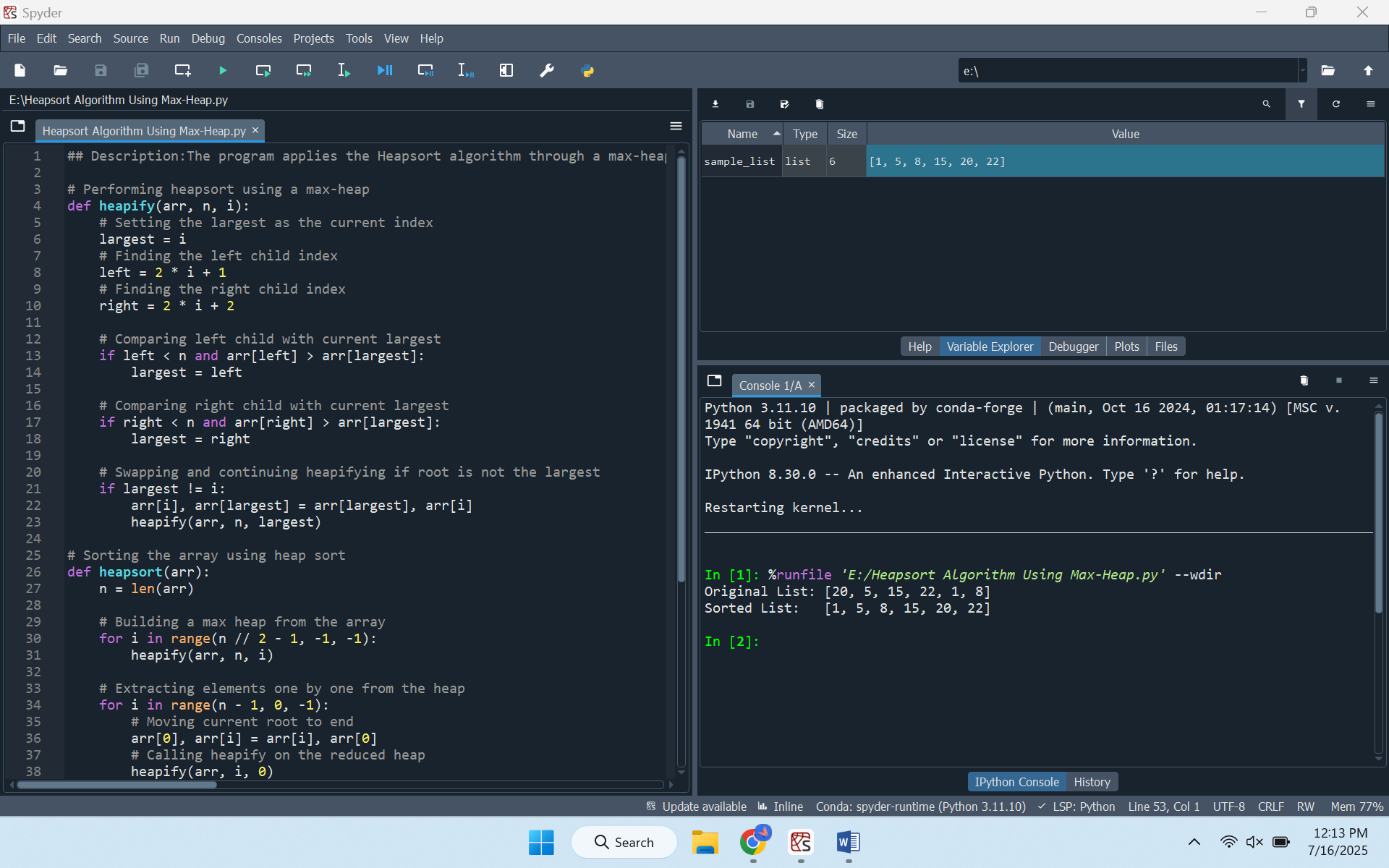
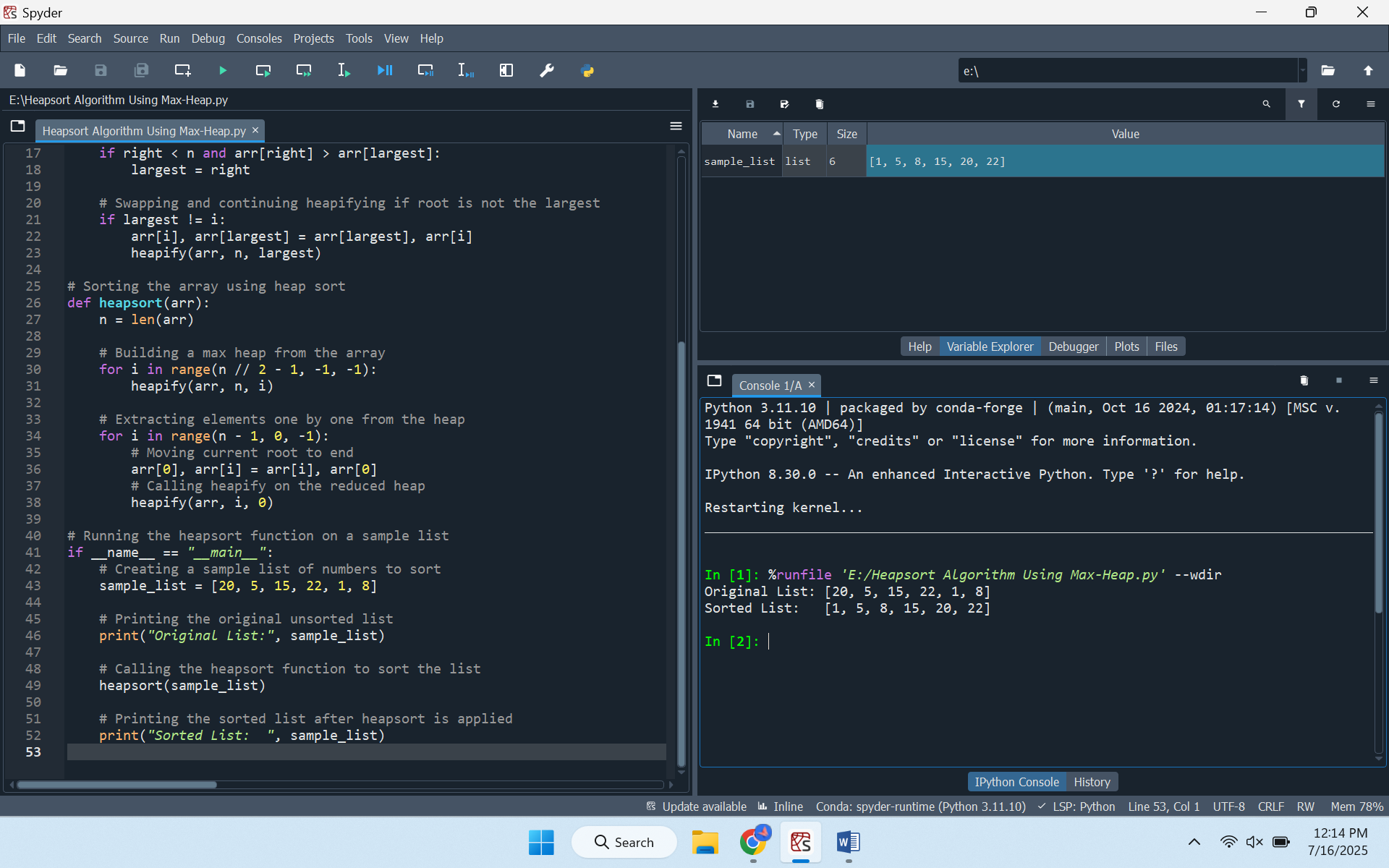
Professor Name

July 16, 2025

**Heapsort Algorithm Implementation**

Heapsort uses a binary max-heap data type where the root node is the largest node. The algorithm starts by transforming the initial array into a max-heap. This is done using the heapify operation, which is applied to each non-leaf node of the array starting at the halfway point of the array and continuing towards the initial point of the array. Once the max-heap is established, the root maximum element is swapped with the last element, and the heap size is decreased by one. Then the heap property is reasserted by using heapify on the new root that is now open. This process will be repeated till the end of the array is sorted in ascending order. The sorting is in-place; thus, no extra memory is needed.

**Heapsort Using Max-Heap**

The Heapsort algorithm resides traditionally through the transformation of the input array into a max-heap and repeatedly removing the maximum element, until the array is put into ascending order. The heap is constructed by heapifying the non-leaf nodes so as to form the initial structure. This is done, and then, the root element and the last element are exchanged, and the obtained sub-heap is again heapified. The procedure is completed in-place, yielding space efficiency while maintaining a worst-case time complexity of O (n logn) for every class of input.  

**Time and Space Complexity Analysis**

Heapsort exhibits uniform time complexity across all three possible cases, best, average, and worst, with each case being O (n logn). This uniformity is a consequence of two principal operations: first, constructing the max-heap in O(n) time, and second, executing each of the n repeated extractions in O (log n) time. The resulting aggregate complexity is therefore O (n log n). The algorithm requires only constant additional space, yielding a space complexity of O (1). Outside of elementary array manipulations that are required to maintain the heap structure, no appreciable overheads are incurred.

**Empirical Comparison with Other Sorting Algorithms**

Heapsort, Quicksort, and Merge sort were compared with relative precision in a comparative empirical study that involved heapsort applied to arrays of various sizes and with different initial configurations-random, sorted, and reverse-sorted (Mohammadagha, 2025). The results indicated that Quicksort delivered the fastest average performance, an advantage attributed to its lower constant factors and effective cache utilization; however, its worst-case complexity without optimization is O(n²). Merge Sort consistently operated at the expected O (n log n) level but demanded additional space owing to its recursive mechanism. Heapsort behaved well in all data distributions, and the measured times were closely matched with the theoretical. Heapsort also found useful application in memory-constrained settings, where sorting in-place remains essential. Together, these results emphasize the strength of Heapsort and its reliability with regard to predictable performance in various data distributions.

**Data Structure Selection and Task Representation**

The concept of the linear array, which is similar to a list representation of data, is chosen for the purpose of building a binary heap because it does not contradict the structural characteristics of the complete binary tree. An array representation allows index-calculable access to both parent and child nodes, which would permit efficient extraction and insertion of heap operations. Also, the array-based structure has a low memory overhead and can be dynamically increased and decreased.

A Task entity is constructed to encompass all task-specific data, including a unique identifier (task ID), a numeric priority value, an arrival timestamp, and a deadline. This sort of object-oriented model is more conceptually clear and can be extended in the future during the scheduling process.

In the case of binary heap, min-heap assignment is implemented, and those tasks, which have the smallest priority values, are assigned as the most urgent and extracted in the first. This arrangement upholds the actions of most scheduling algorithms which favor those tasks with the earliest deadline or ones with the greatest urgency.

**Insert Operation and Time Complexity**

The insert operation adds the new task at the back of the heap array and performs the upward traversal to restore the heap property. At this traversal, the inserted new task is compared with its controlling task, and any necessary swap is executed until there is an appropriate location in place. The time complexity of this operation is O (log n), as the maximum number of swaps is proportional to the height of the heap.

**Extract-Min Operation and Time Complexity**

The extract-min operation deletes and reveals the least important task at the top of the heap. The last item of the heap is swapped with the root, and the further downwards heapify procedure is then used to rebuild the heap structure. The procedure ensures that the next task with maximum priority is taken out in the next extract-min operation. Moreover, the time complexity of extract-min remains O (log n), as, at most, the height of the binary heap is examined during the heapify phase.

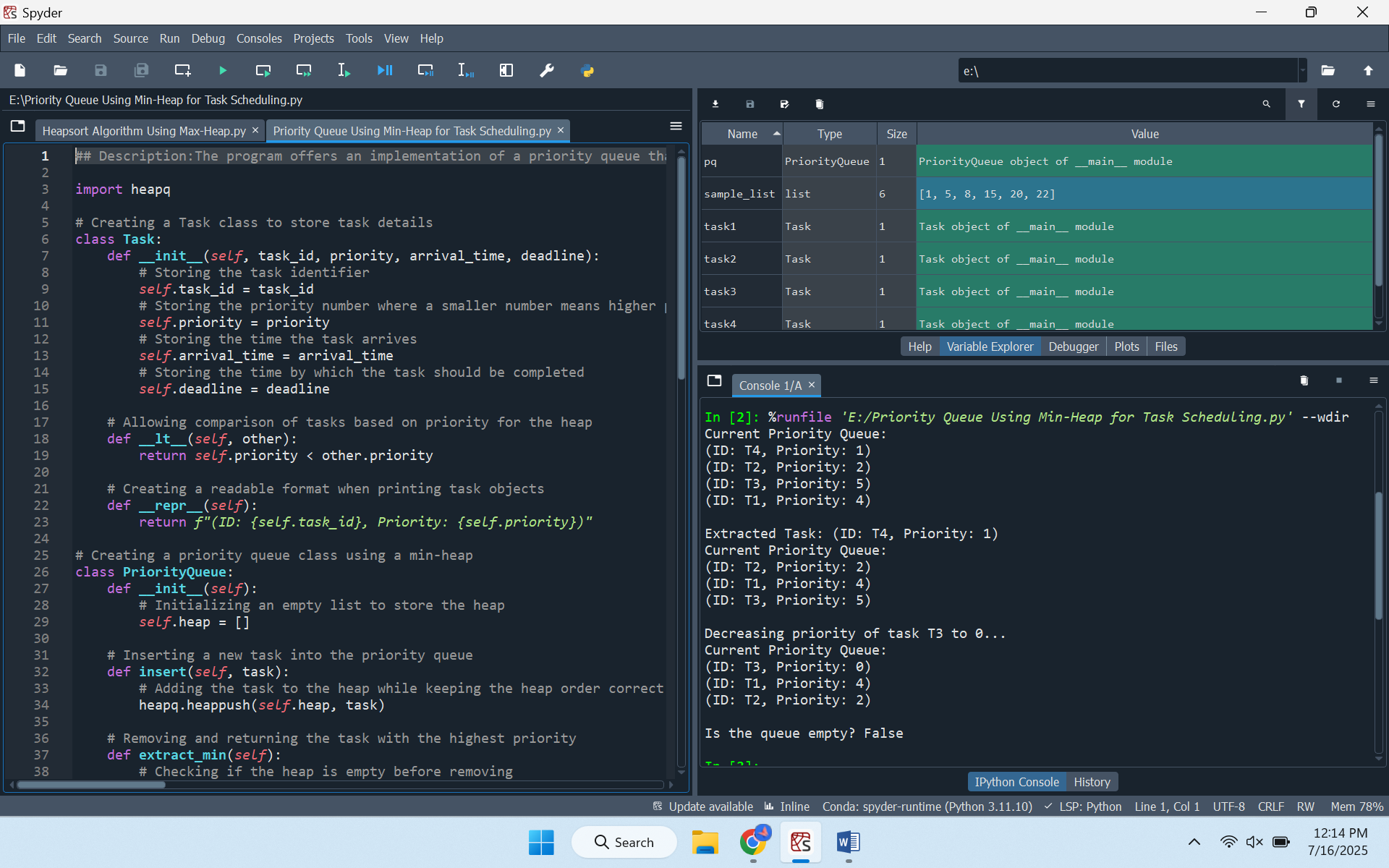
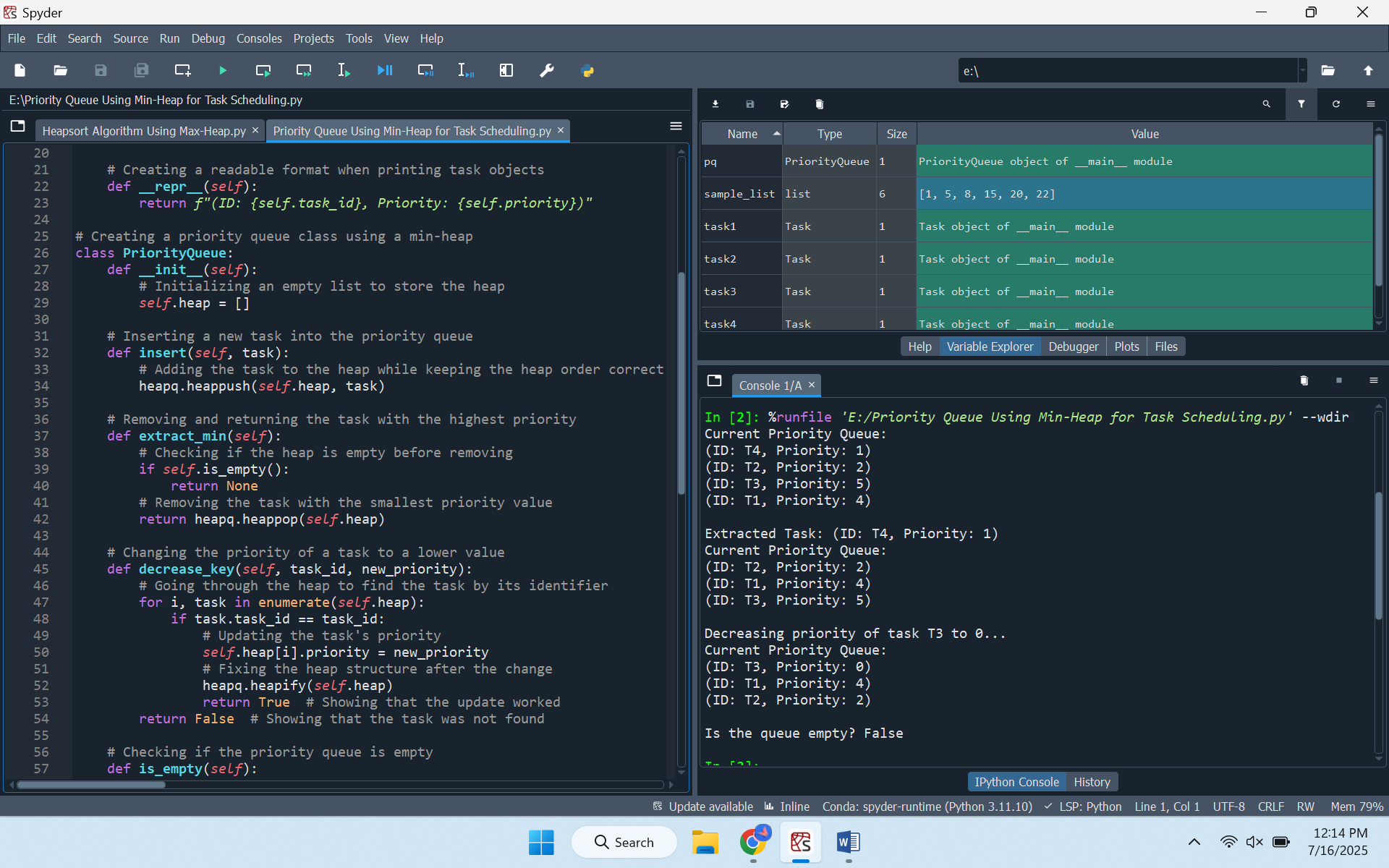
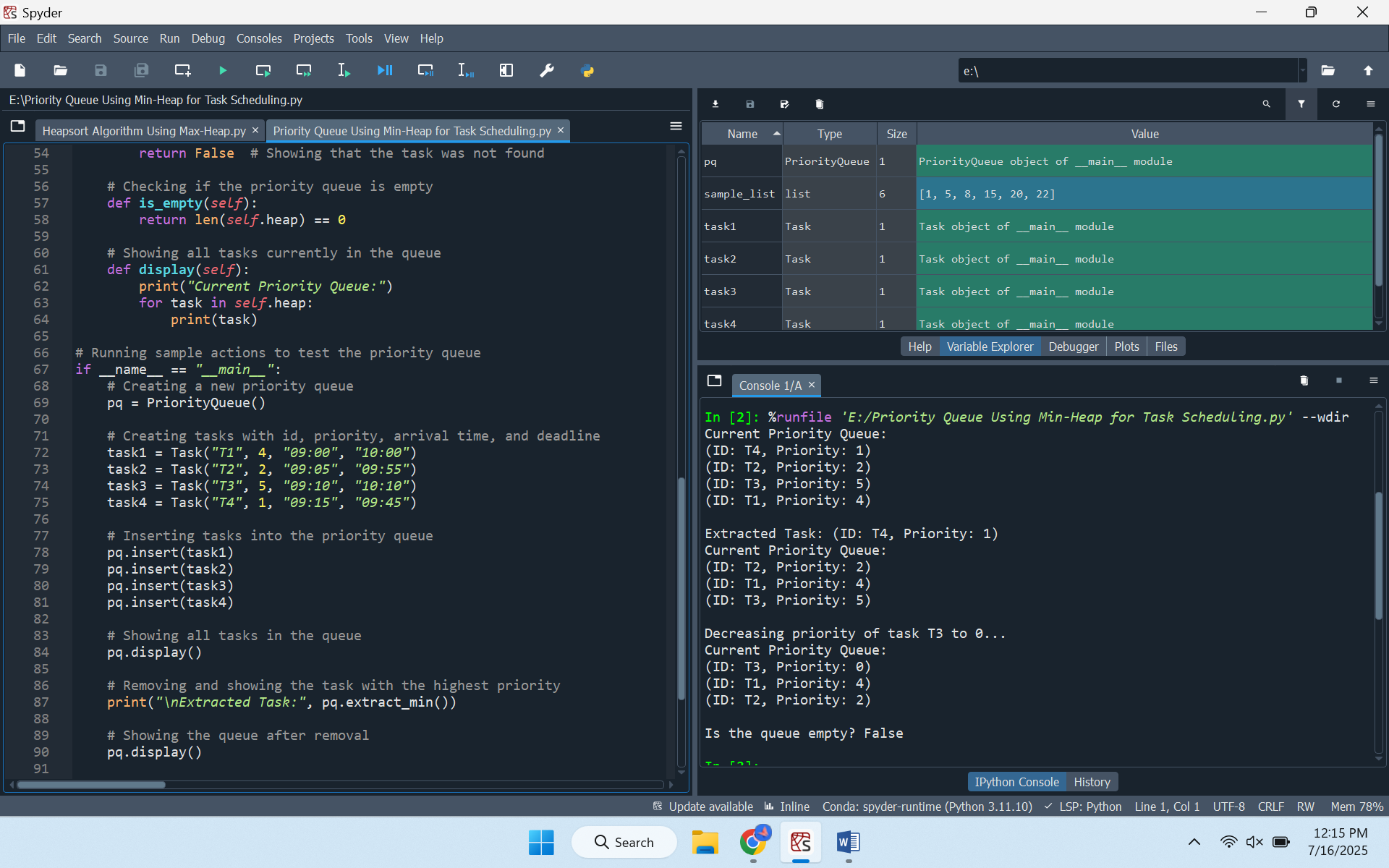
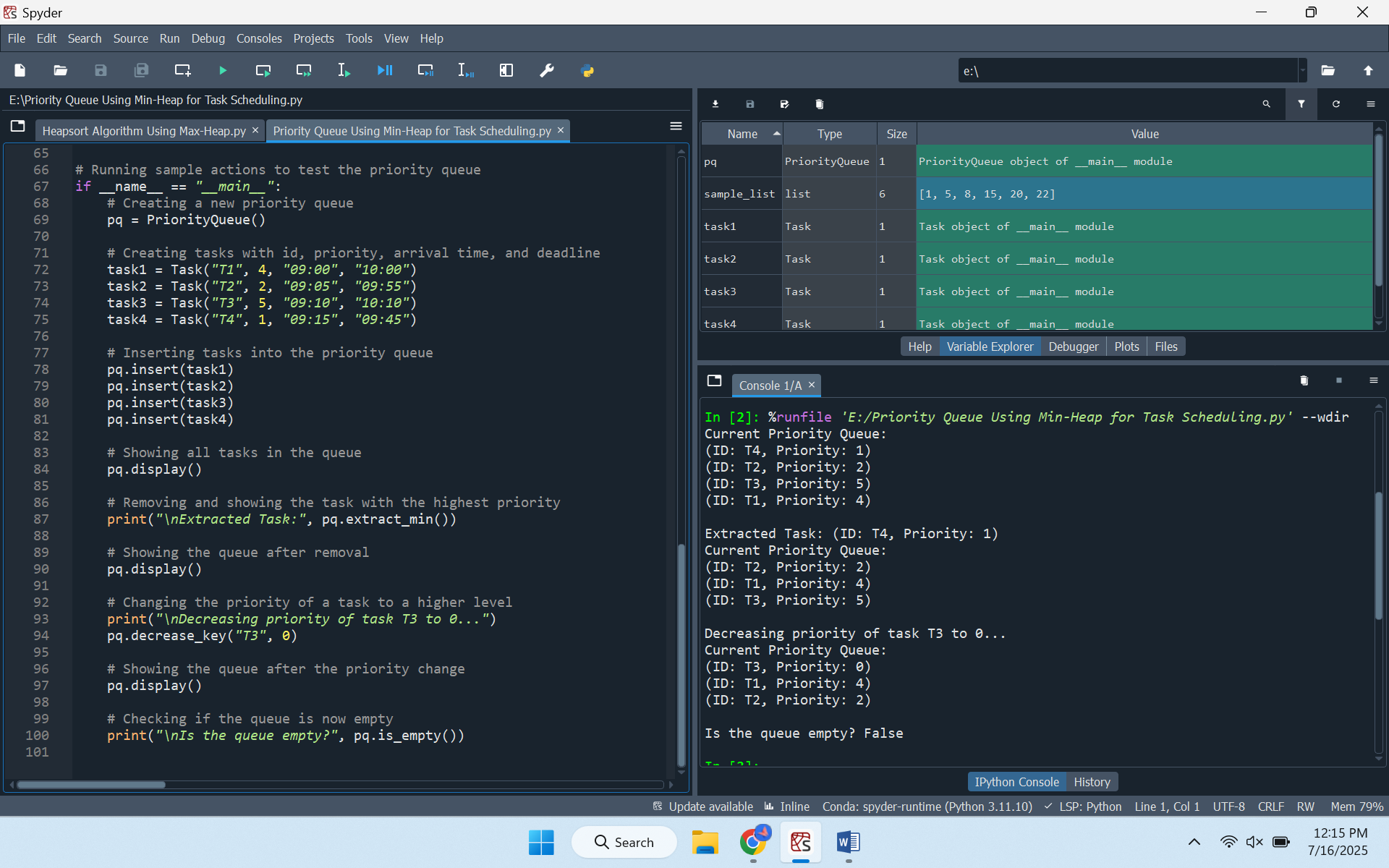
**Decrease-Key Operation and Time Complexity**

The decrease-key operation alters the priority of an existing task and assigns it smaller value, increasing the urgency of the task. Following this adjustment, the heap property is reinstated via upside-down traversing the adjusted node and exchanging where vital. In an array-based heap lacking direct indexing support, identifying the task by its ID engenders a linear scan, yielding time complexity O(n) for the search phase and O (log n) for the adjustment, producing an overall O(n) operation.

**Is-Empty Check and Time Complexity**

The is-empty operation determines whether a heap array contains elements, thereby requiring only a single traversal to determine its status, owing to this constant-time complexity, the operation is classified as O (1). This quick assurance is also vital to the effective task-queue processing by scheduling systems where queue overheads must be minimized.

**Priority Queue Using Min-Heap for Task Scheduling**

A priority queue is typically implemented as a min-heap data structure, with tasks sorted on a priority value: the lower a value, the more important (Benomar & Coester, 2024). To ensure reasonable scheduling behavior, the structure should be able to support four basic operations: insertion of a new task, extraction of the most urgent task, reduction of priority of a current task, and empty queue checking. Internally, tasks are modeled using their attributes, whose values include identifier, priority level, arrival time, and deadline. The heap property is maintained throughout each operation, which in turn guarantees optimal performance of the schedule.    

**Conclusion**

Heapsort and priority queues in the form of heap data structures are both practically efficient and theoretically solid as an example of the study of computer algorithms, as shown by an empirical study. Heapsort has quality results with its predictable time that remains constant and minimal space needs, advantages that make it appealing in memory-intensive platforms. Min-heaps are utilized to ensure priority queues are obtained that enable tasks to be scheduled efficiently by choosing to process the most urgent task at all times. In combination, these structures demonstrate the effectiveness of heaps as an application to real-world problems that focus on order, priority, and efficient computation. The consistency of their action and in favor of dynamic activity makes them suitable instruments worthy of both scholarship and practice.

**References**

Benomar, Z., & Coester, C. (2024). Learning-augmented priority queues. Advances in Neural Information Processing Systems, 37, 124163-124197.

Mohammadagha, M. (2025). Hybridization and Optimization Modeling, Analysis, and Comparative Study of Sorting Algorithms: Adaptive Techniques, Parallelization, for Mergesort, Heapsort, Quicksort, Insertion Sort, Selection Sort, and Bubble Sort.