**Assignment 6: Medians and Order Statistics & Elementary Data Structures**

Student Name

University

Algorithms and Data Structures - Bi-term2

Professor Name

July 29, 2025

**Introduction**

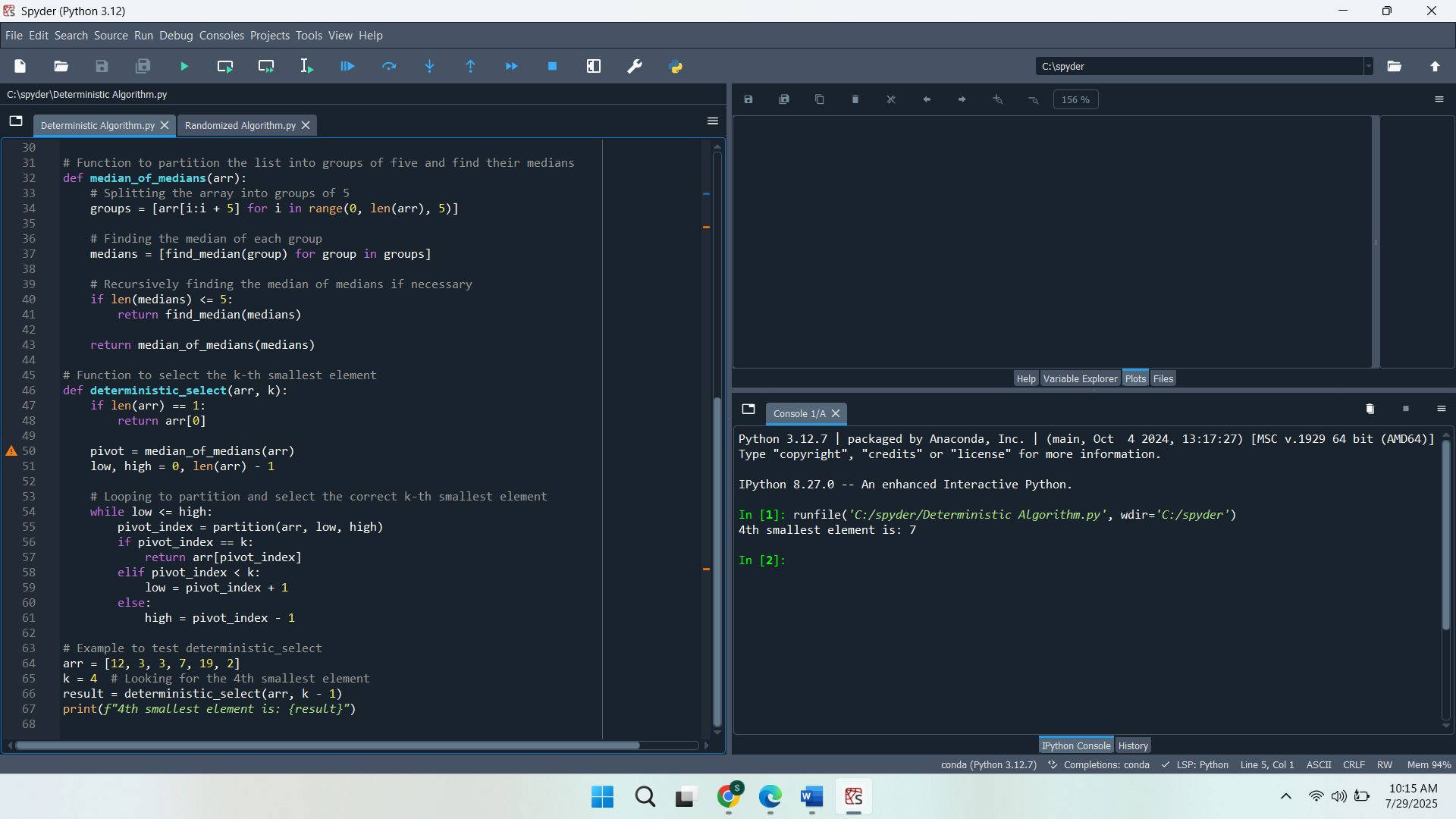
The aim is to consider significant issues, including medians, order statistics, and low-level data organizations. These are used extensively in such fields as data analysis, optimization, and computational geometry. The Median of Medians algorithm is one of these methods, and it ensures a linear-time worst-case search for the k-th smallest element of an unsorted array. Randomized Quick Select is another method that can be employed to give expected linear time results by randomly choosing the pivot. Through the analysis of the two algorithms, it is more likely to determine their efficiency and time complexity of the same. It is concerned more with finding out how these algorithms operate and how they could be used successfully in a practical situation.

**Part 1: Implementation and Analysis of Selection Algorithms**

**1. Implementation**

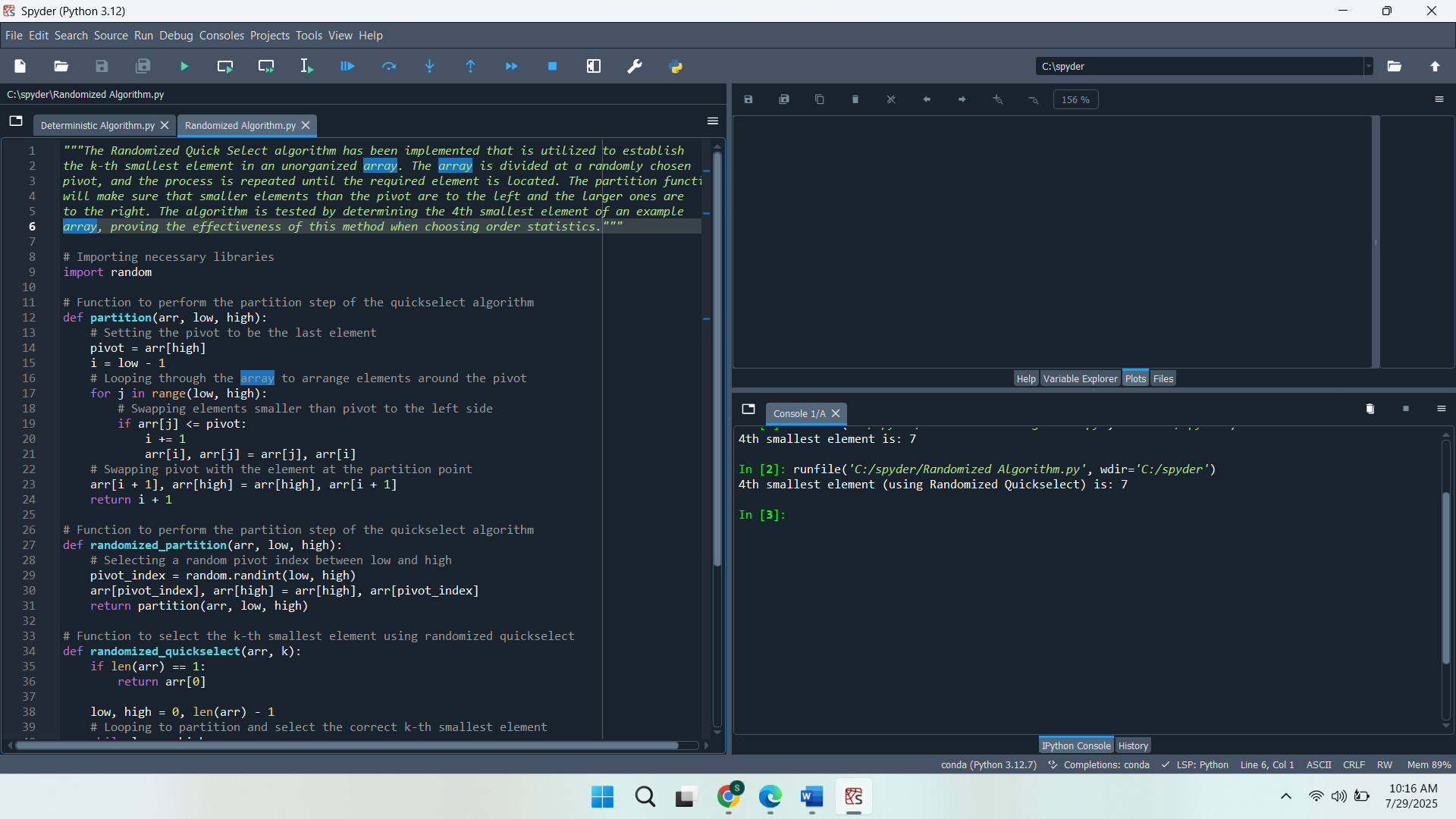
**Deterministic algorithm**

Using the Median of Medians algorithm to choose the k-th element of an unordered array of numbers. It splits the array into roughly two parts around a pivot, and then recursively selects a median of medians used as a pivot for efficient selection. The algorithm also has worst-case linear time. In the example, it finds the 4th smallest element by calling the deterministic\_select function with a sample array, demonstrating how the algorithm works for selecting order statistics.



**Randomized algorithm**

The Randomized Quick Select algorithm has been implemented that is utilized to establish the k-th smallest element in an unorganized array. The array is divided at a randomly chosen pivot, and the process is repeated until the required element is located. The partition function will make sure that smaller elements than the pivot are to the left and the larger ones are to the right. The algorithm is tested by determining the 4th smallest element of an example array, proving the effectiveness of this method when choosing order statistics.



**2. Performance Analysis**

**Time Complexity Analysis for Both Deterministic and Randomized Selection Algorithms**

The deterministic selection algorithm guarantees a worst-case time complexity of O(n) due to its methodical partitioning and pivot selection. The array is separated into small sets, and the median of the set is chosen. This also has the property of picking the pivot to be in the middle of the array, since the median of medians is always near the middle of the array, so each partition will be balanced. This ensures that the problem size reduces effectively, leading to a time complexity of O(n), as the recurrence relation T(n)=T(n/5) O (n) simplifies to O(n).

The randomized selection algorithm achieves an expected time complexity of O(n). The pivot chosen randomly creates a divide in the array, which, in general form gives a balanced subarray. This cutting of the size of the problem by a factor of 2 at each step also leads to the anticipated linear time complexity. The partitioning step takes O(n), and on average, the problem size reduces by half with each recursive call. However, in the worst case, where the pivot selection results in unbalanced partitions, the time complexity degrades to O(n2). This notwithstanding, the time complexity is still expected to be linear because the pivot being chosen is random.

**Deterministic Algorithm Achieves O(n) Time Complexity in the Worst Case, While the Randomized Algorithm Achieves O(n) Time Complexity in Expectation.**

The deterministic algorithm achieves O(n) time complexity in the worst case due to its structured approach to pivot selection. The algorithm makes a good choice of the pivot by dividing the array into groups and by choosing the median of the medians, which leads to a good balance of partitions. This prevents the case of worst partitioning, which is nonlinear in the worst case. The recurrence relation reflects the consistent reduction in problem size, making the worst-case time complexity O(n).

The randomized algorithm achieves O(n) expected time complexity because the random pivot selection typically divides the array into two roughly equal subarrays, leading to efficient partitioning. Each recursive call cuts the problem size in half, which gives the expected time complexity of linear size. However, the algorithm can experience worst-case behaviour when the pivot consistently divides the array poorly, leading to unbalanced partitions and a time complexity of O(n2). Nonetheless, the anticipated case is linear because of the random selection of the pivot.

**3. Empirical Analysis**

**Empirical Comparison of Running Times**

Comparing the performance of deterministic and randomized algorithms of selection of different distributions of inputs, as random, sorted, and reverse-sorted arrays of arrays, some peculiarities are observed:

* **Random Arrays:**

In the case of random arrays, the two algorithms are likely to behave in similar ways. The deterministic algorithm guarantees a worst-case time complexity of O(n), while the randomized algorithm typically performs in O(n) on average. Both algorithms divide the array around a pivot, and the running time difference between the two algorithms is negligible on moderately sized inputs. Nonetheless, determinism gives more predictive performance.

* **Sorted Arrays:**

The randomized algorithm might not perform quite well on sorted arrays. The random selection of the pivots may result in skewed partitions, and this may result in increased recursive calls. In contrast, the deterministic algorithm consistently achieves O(n) performance because its pivot selection ensures balanced partitions, even with sorted input.

* **Reverse-Sorted Arrays:**

On a reverse-sorted array, the randomized algorithm can be expected to perform as well or superior to a sorted array, because the random pivot eliminates the effects of a succession of unrepresentative partitions. Nevertheless, it can perform a little differently. The deterministic algorithm continues to perform consistently in O(n) time, regardless of the input order.

**Observed Results vs. Theoretical Analysis**

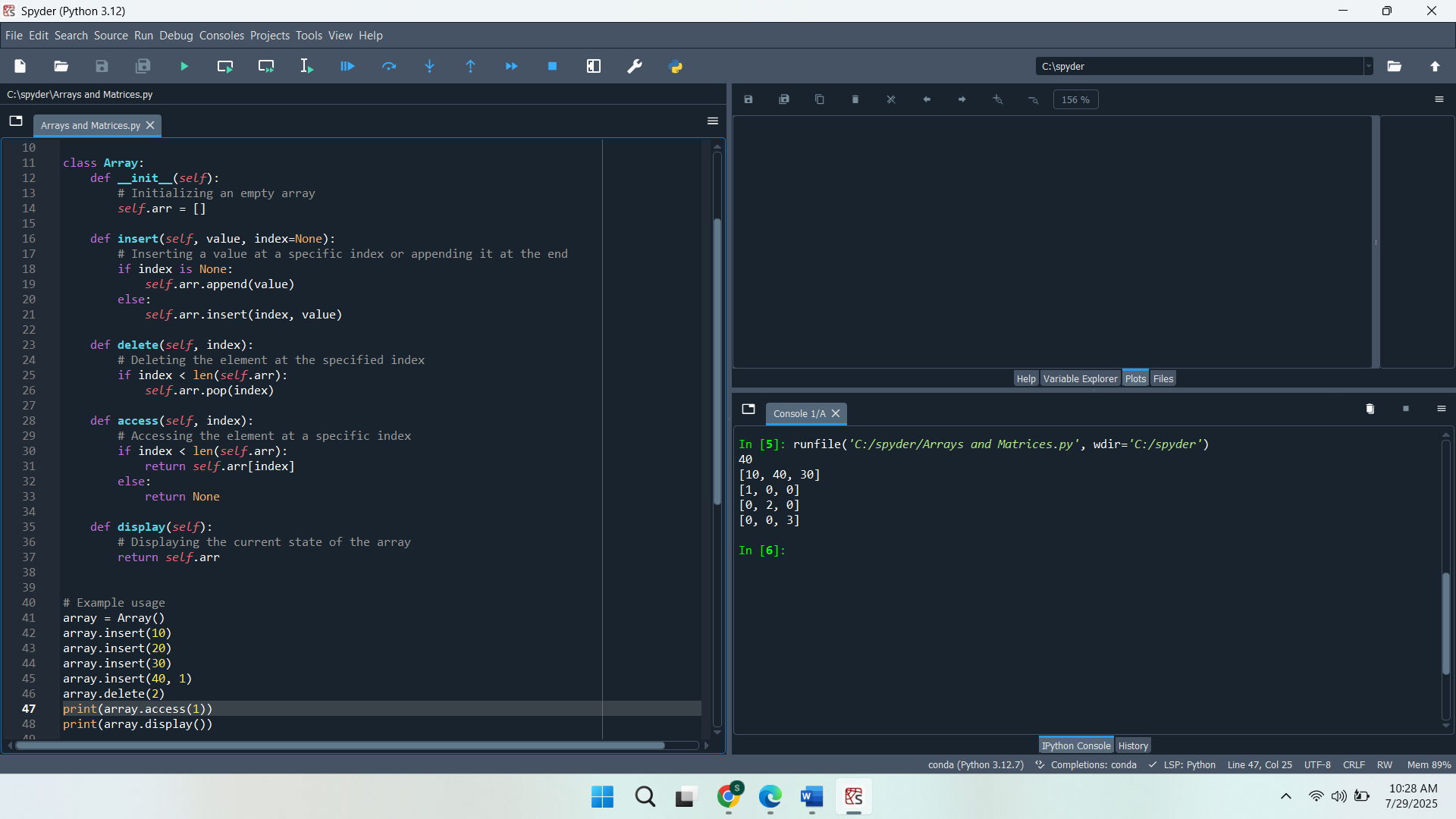
The empirical tests provide comparative results as compared to the theoretical time complexities of the two algorithms. The deterministic algorithm is quite reliable in its performance with deterministic running times, particularly on sorted or reverse-sorted data, where its pivot selection method gives balanced partitioning. The randomized algorithm, while expected to perform in O(n) time on average, showed some fluctuations in its running time, particularly with sorted or reverse-sorted arrays, where poor pivot selections could lead to unbalanced partitions. In general, the deterministic algorithm is much steadier and regular, whereas the randomized algorithm is more effective and usually works efficiently under the majority of random conditions.

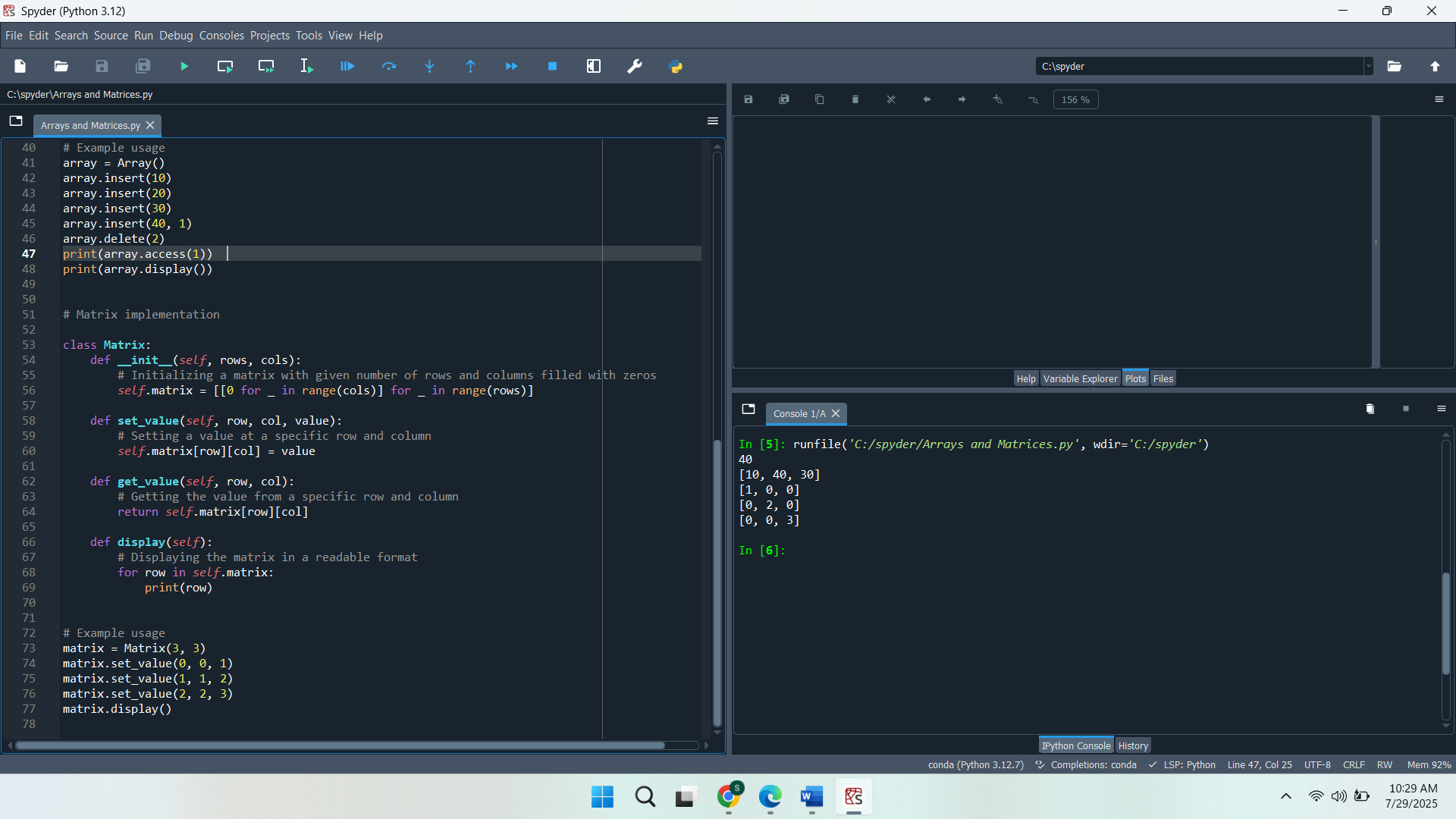
**Part 2: Elementary Data Structures Implementation and Discussion**

**1. Implementation**

**Arrays and Matrices**

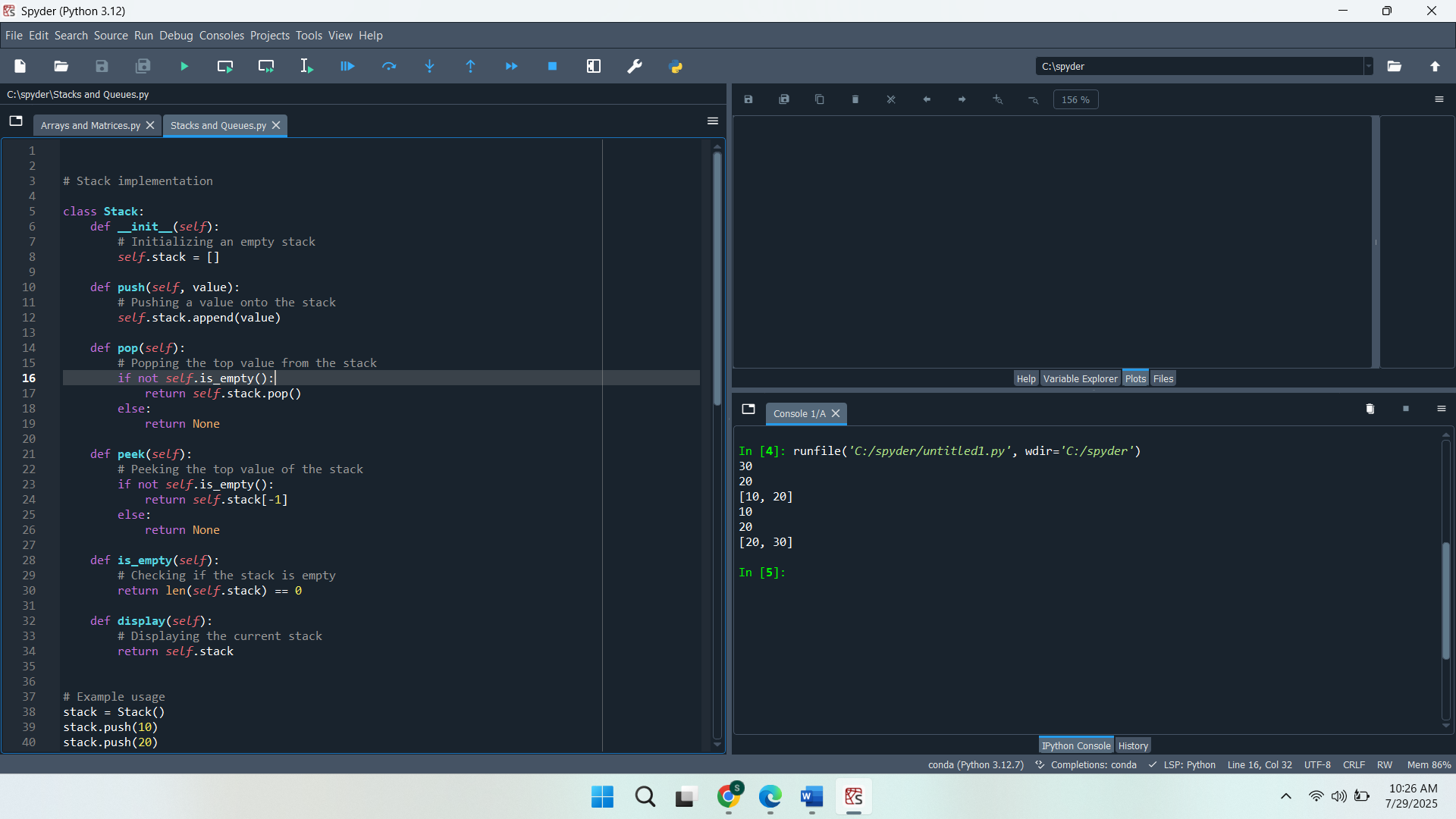
Declaring two classes as Array and Matrix. The Array class defines such simple methods as insertion, deletion, access, and display to work with the array. The Matrix class models a two-dimensional matrix and implements set, get, and print methods to set, get, and print matrix values. In the sample usage, an array is used to insert, delete, and access elements in the array, whereas the matrix is made, the elements in the matrix are filled, and also shown. The two classes apply simple list-based implementations to carry out their functions.

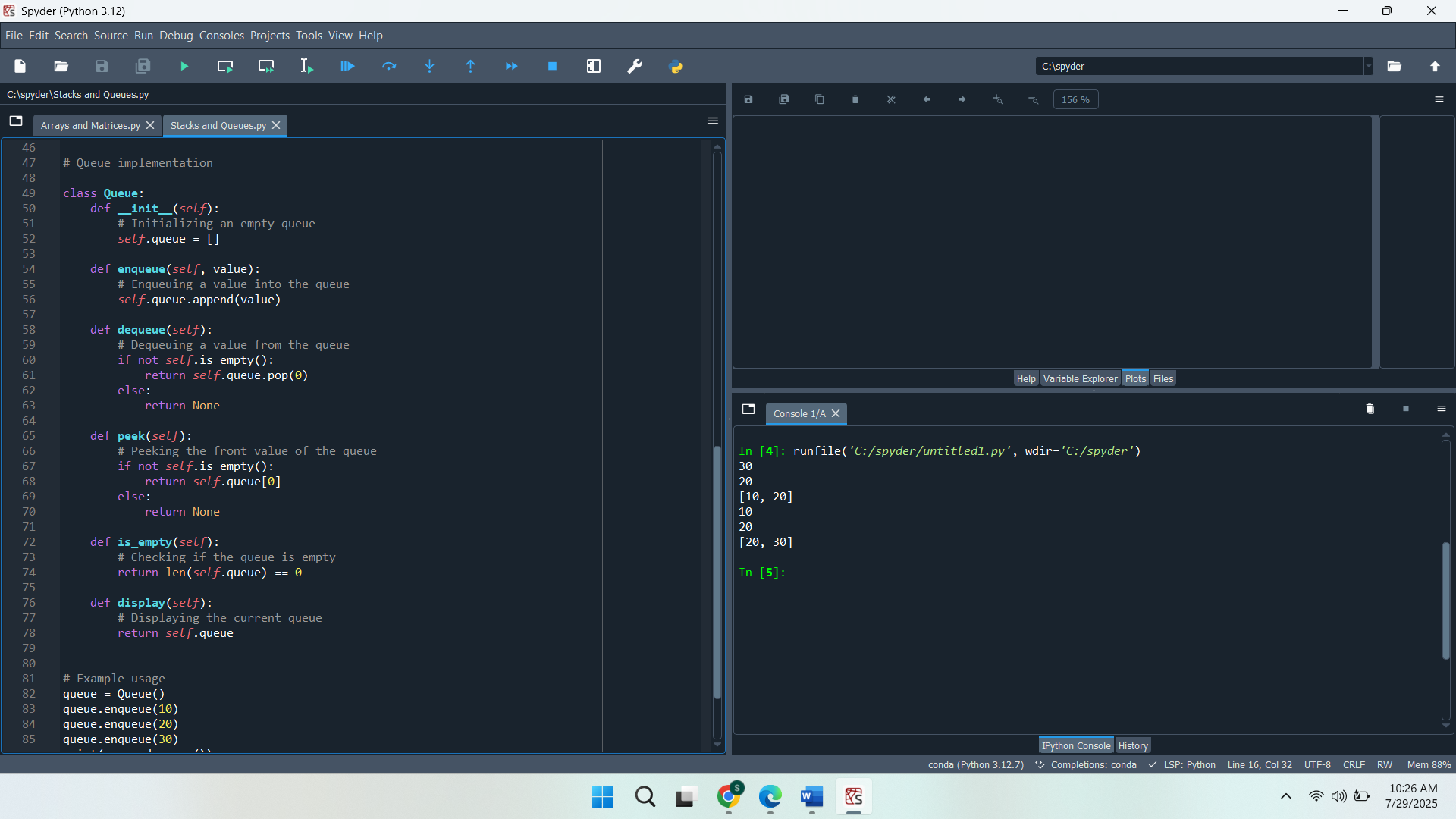




**Stacks and Queues**

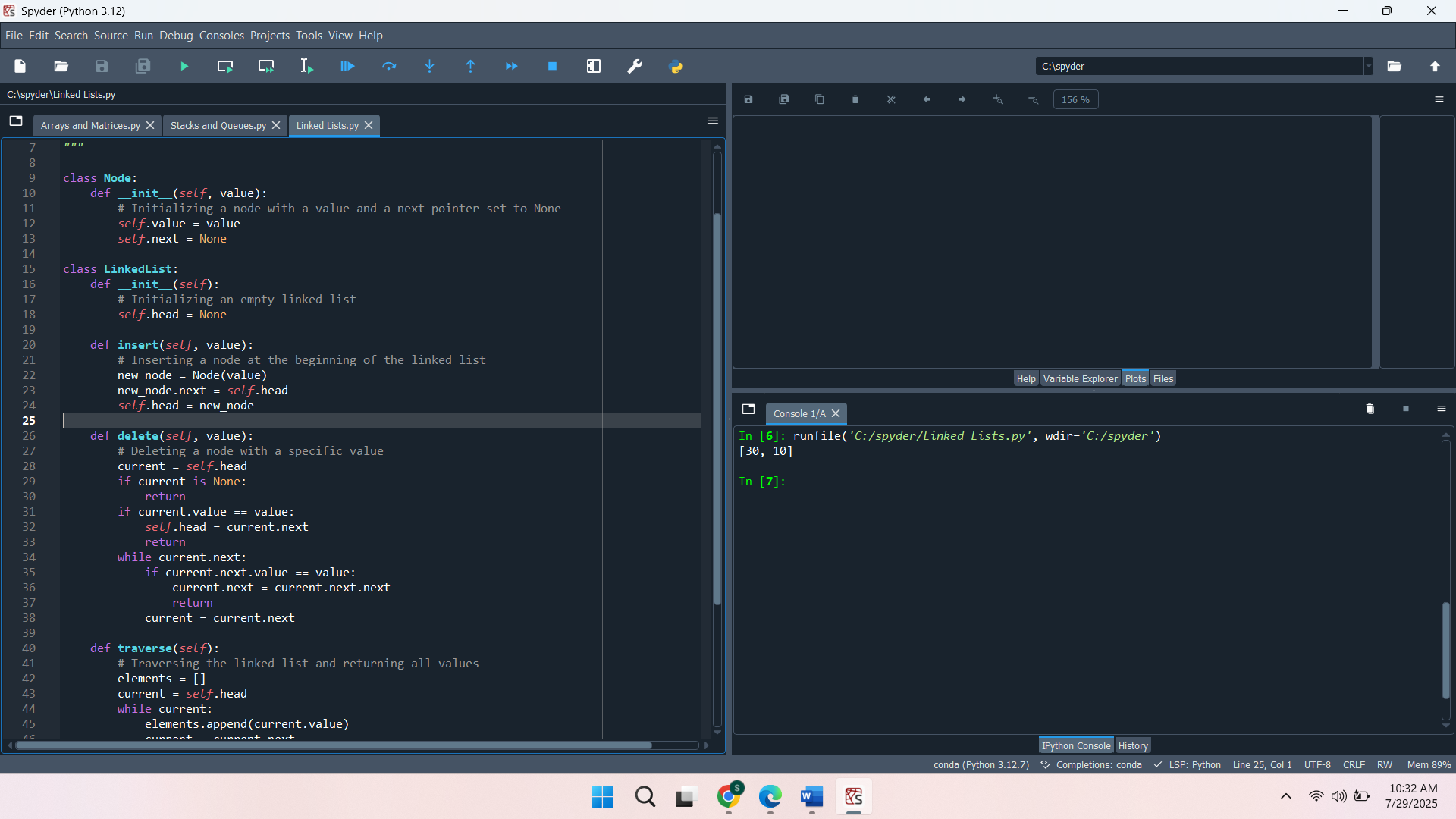
Elaborate on two classes named Stack and Queue, and implement the operation of each of them. The stack class implements methods to push, pop, peek the stack, empty stack check and print the stack. Queue provides enqueueing, dequeuing, peeking, checking empty status, and printing the queue. In the example usage, both data structures are used by adding, removing, and accessing elements to illustrate their behaviour. Both classes have an efficient data handling approach through the use of list-based implementation in Python.

****

****

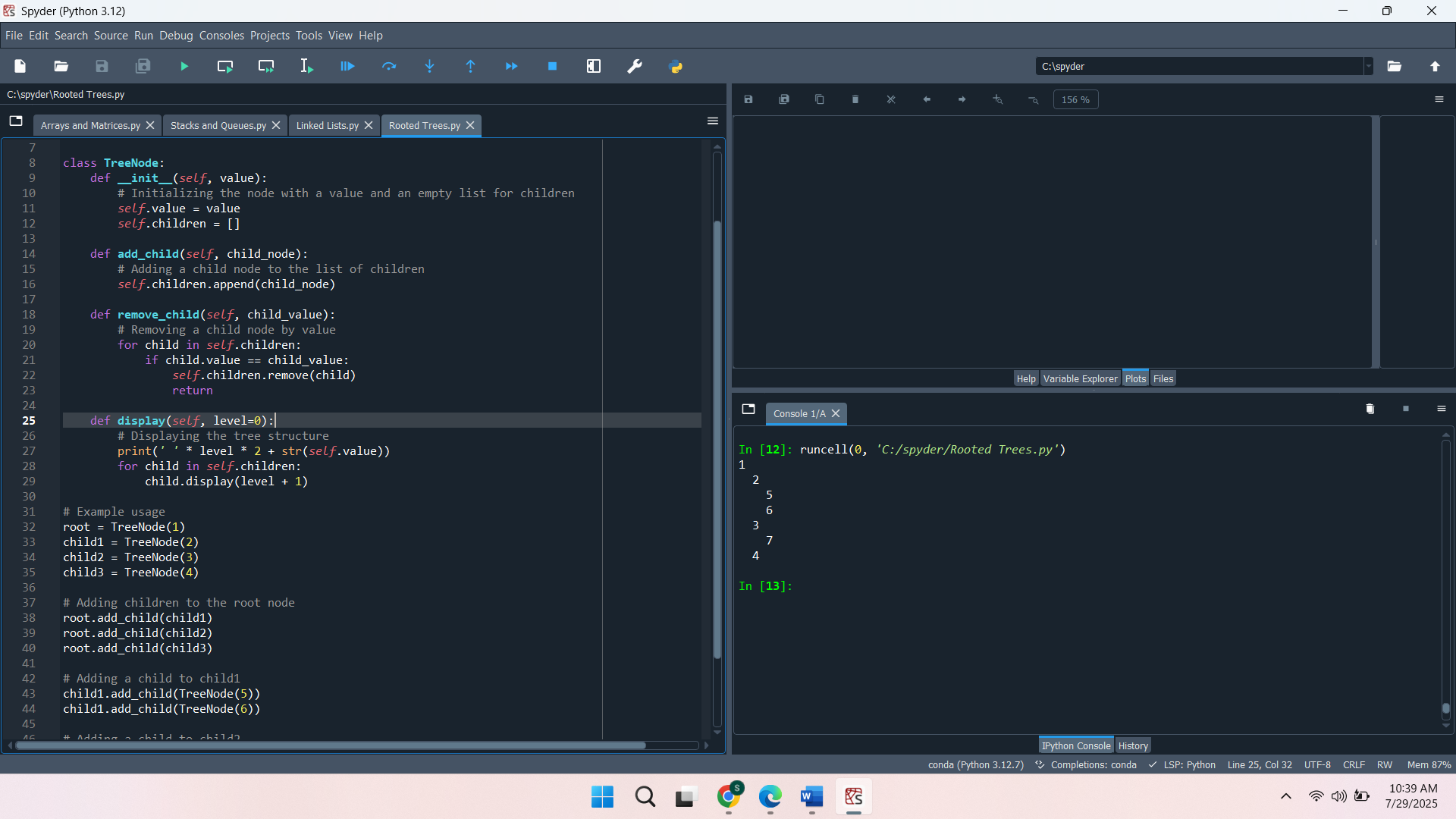
**Linked Lists**

Implementing a linked list class with some simple methods like inserting, deleting, and traversing. Each of the elements of the linked list is given as an instance of a Node class, which can include a value and a reference to the next node. The insert method inserts nodes at the start, delete method deletes a node whose value is given. The traverse method provides the entire values of a linked list in the form of a list. The example illustrates inserting, deleting, and traversing the linked list.



**Rooted Trees**

Declaring a TreeNode that deals with each node in a rooted tree, whose nodes carry a value and a collection of children. The add\_child method allows adding child nodes, while the remove\_child method removes a specific child by value. The printout is used to display the hierarchy of nodes by printing the tree structure using indentation recursively. The sample code creates a tree of a root node and multiple child nodes, followed by the representation of the tree structure.

****

**2. Performance Analysis**

**Time Complexity for Basic Operations**

The time complexity of simple operations in the array and linked list is quite different because of the different structures. Arrays provide O(1) time complexity for element access and insertion at the end, but operations like insertion or deletion at arbitrary positions take O(n), as elements need to be shifted. Linked lists, on the other hand, offer O(1) time complexity for insertion and deletion at the beginning (or end if a tail pointer is maintained), but accessing elements requires O(n) time as the list must be traversed from the head to the desired node. Stack and queue operations, such as push, pop, enqueue, and dequeue, all take O(1) time for both arrays and linked lists, with arrays having potential resizing overhead for dynamic implementations.

**Trade-offs Between Using Arrays and Linked Lists for Stacks and Queues**

Arrays and linked lists can both be used to implement stacks and queues, depending on the unique application. Arrays are space-efficient, with O(1) time complexity for access and appending elements at the end, making them ideal for applications with a known and fixed size. But array resizing may come with a lot of overhead, especially when resizing dynamically. Linked lists, in contrast, are more flexible, enabling dynamic setting and removal of size without resizing. Insertions and deletions are efficient at the beginning and end of the list, but accessing an element requires O(n) time, making them less suitable for applications that require frequent random access. Using both data structures, stacks and queues may be implemented, although linked lists are frequently used when the structure size is changing or when many insertions and deletions are required.

**Efficiency of Different Data Structures in Specific Scenarios**

There are different advantages that various data structures present depending on the requirements. The most effective usage of arrays is when speed of access to the elements is important, particularly in applications where data is not dynamic or knowledge of the size is known beforehand, like in a look-up table or dynamic programming. They are also offered in a way that is suited to applications that involve many reads and fewer updates. Linked lists are good in cases where dynamic data handling is necessary, like in real-time data storage, wherein there are many insertions and deletions, and the data size is not predetermined. A linked list will be better suited where dynamic memory usage is needed and where elements must be added and removed often, with no need for random access performance. Linked lists tend to perform better than stacks and queues where insertions and removals are involved in the queue, such as the dequeue operation. Otherwise, an array would be shifting elements. Arrays are, however, easier to realise and potentially perform better on static or smaller applications.

**3. Discussion**

**Practical Applications of Data Structures in Real-World Scenarios**

Linked lists, arrays, queues, and stacks are all very important as part of real-world computing environments. The data structures appear in numerous, diverse applications, where the selection of structure affects performance, efficiency, and scalability directly.

**Arrays:**

Arrays are used practically in most applications since they use quick indexing and fixed memory. Arrays are very useful when the size of the data is known a priori, and random access must be achieved in moving image processing, matrix computation, and dynamic programming. They are also highly effective in applications like sorting algorithms and caching mechanisms, where fast access to elements is essential. Array is, however, not appropriate when dynamically sized data or when insertions and deletions are too imminent, as it is fixed in size and requires shifting of the elements.

**Linked Lists:**

Linked lists are especially practical in programs where data is dynamic or dynamically modified. They are frequently applied when dynamically allocating memory, in the memory management system of the operating system, or in a garbage collection system, where memory can be dynamically retrieved and returned. Linked lists can also be used in real-time data processing when data comes in an arbitrary size. Their capacity to insert or remove elements with ease and non-displacement of the other elements makes them the preferred option in cases involving dynamic datasets that change in size, whether enlarging or reducing.

**Stacks:**

Stacks are fundamental in applications where a last-in, first-out (LIFO) order is necessary. They are heavily used in function call management in programming languages, where each function call pushes an activation record onto the stack, and once the function completes, it is popped from the stack. Stacks also find applications in undo mechanisms in software, depth-first search (DFS) in graph traversal, and parentheses matching in compilers. In such situations, stacks are favoured because of their easy implementation and handling of nested operations.

**Queues:**

Queues are optimal for first-in, first-out (FIFO) processes and are widely used in task scheduling, such as in operating systems, where processes are queued for execution. Message handling systems also use them. Additionally, breadth-first search (BFS) in graph traversal employs queues to ensure nodes are visited in the correct order. Queues are essential in scenarios where maintaining the order of operations is critical, such as in real-time data streaming applications, customer service systems, and load balancing systems.

**Scenarios Where One Data Structure May Be Preferred Over Another**

The choice of one data structure and another greatly depends on the memory consumption and speed, or user friendliness of the code to be used. Those are the factors that can determine which of the arrays, linked lists, stacks, and queues to utilize.

* **Memory Usage:**

In situations where the size of the data is known and fixed, arrays are memory efficient. They are, however waste of memory when the size of allocated memory is far bigger than necessary. Differently, however, the linked lists' memory allocation is dynamic, it can expand and contract according to demands, and as a result, they are more memory-efficient in situations when the amount of data to manage may not be predictable.

Linked lists have overhead because each node will have a pointer, and this can be a disadvantage when dealing with a lot of elements. In cases where memory capacity is not a concern or data is small, arrays can be used because of their ability to consume the suppressed space.

* **Speed:**

Arrays provide fast random access (O(1)) to elements, making them ideal when frequent access to specific elements is required. However, their insertion and deletion operations can be slow (O(n)) when they involve shifting elements.

Linked lists, on the other hand, excel in insertion and deletion operations O(1) at the head or tail but suffer from slow access time (O(n)) since each element must be accessed sequentially.

Stacks and queues, when implemented with arrays, offer O(1) push and pop operations, which makes them fast in scenarios requiring a simple, sequential flow of data. When implemented with linked lists, both structures still offer O(1) operations, but with the added overhead of memory for the pointers.

* **Ease of Implementation:**

Arrays are readily implemented and simple to process and manage a small, fixed-sized set of data. They are appropriate to such problems when the size is already known and sorting or searching is a common operation.

Linked lists are more procedural than arrays are because they involve more complicated code at the expense of handling the pointers of each node. They do, however, come with the flexibility required in dynamic data structures.

Stacks and queues are easily implemented, particularly using arrays, and are found in most algorithmic applications such as recursion, scheduling, and resource management.