Week: 11.1 AI ASSIST

2303A52281

T. Vikas

TASK-1

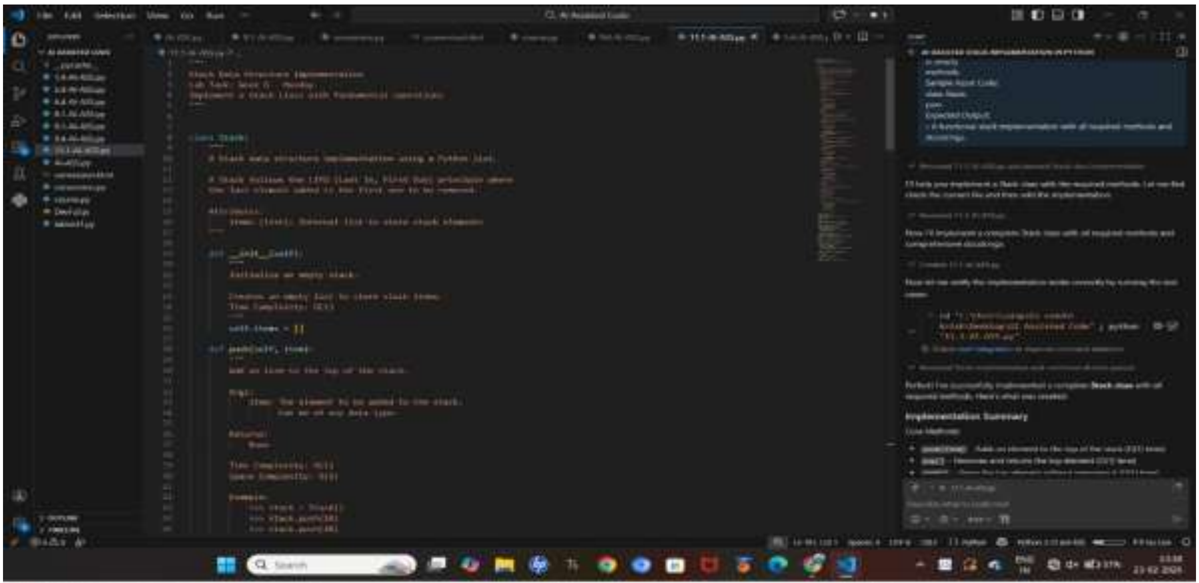
Scenario:

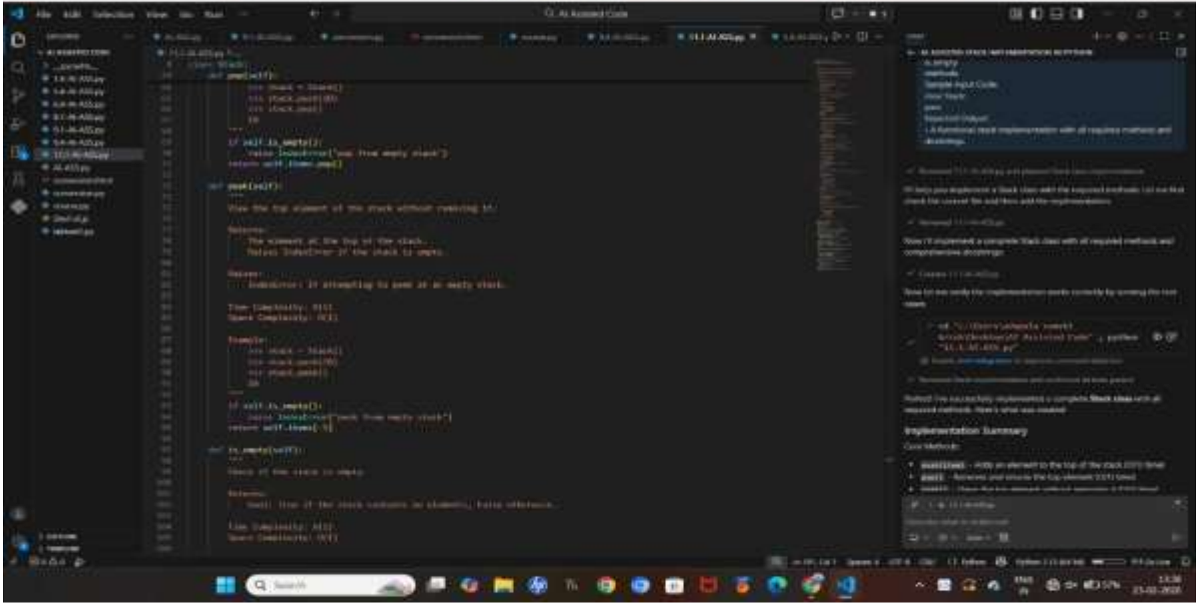
In this task, we used AI to design and implement a Stack data structure in Python. The objective was to understand how a stack follows the LIFO (Last In, First Out) principle and how AI can assist in writing structured, optimized, and well-documented code. Instead of manually implementing the class, AI was prompted to generate a complete and functional Stack with essential operations.

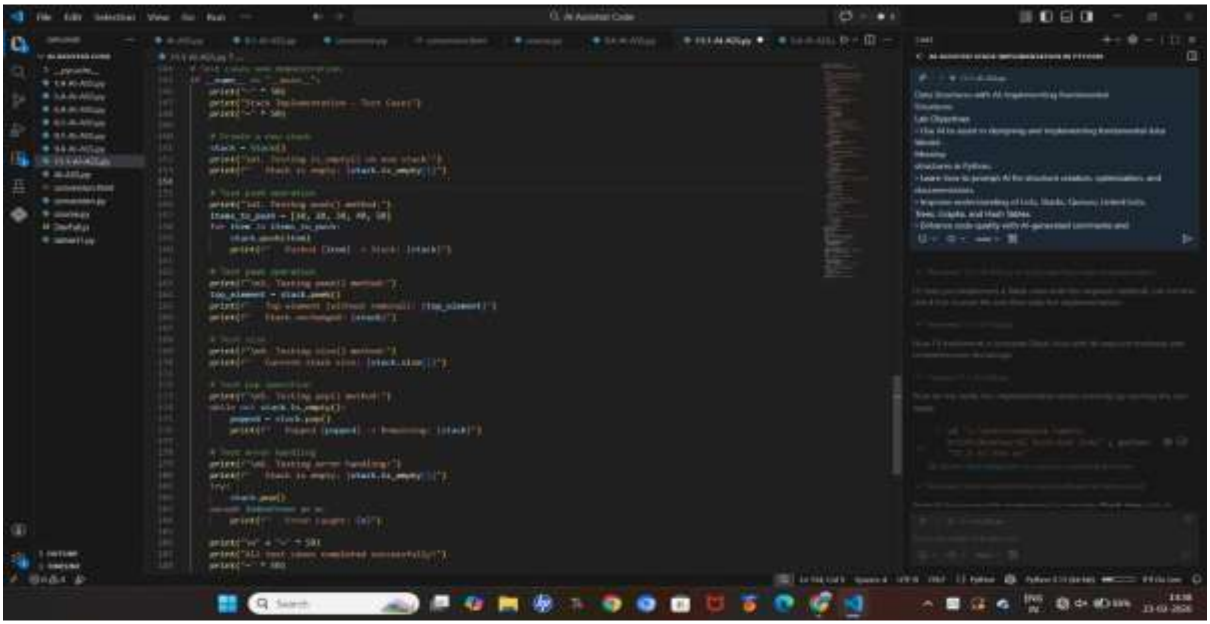
Prompt:

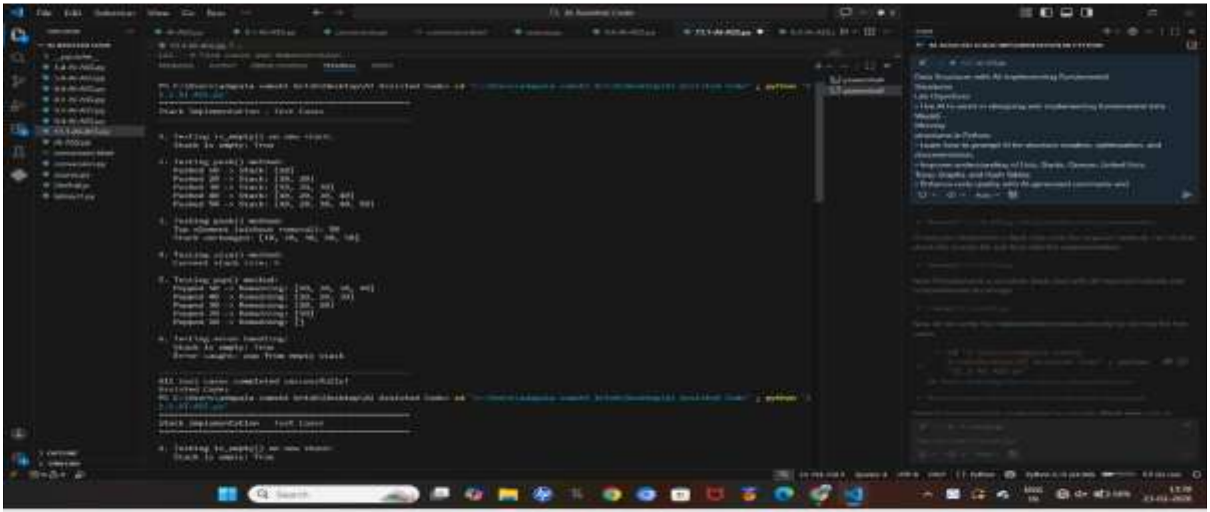
“Generate a beginner-friendly Python Stack class using a list. Include push, pop, peek, and is_empty methods. Add proper docstrings and handle edge cases like popping from an empty stack.”

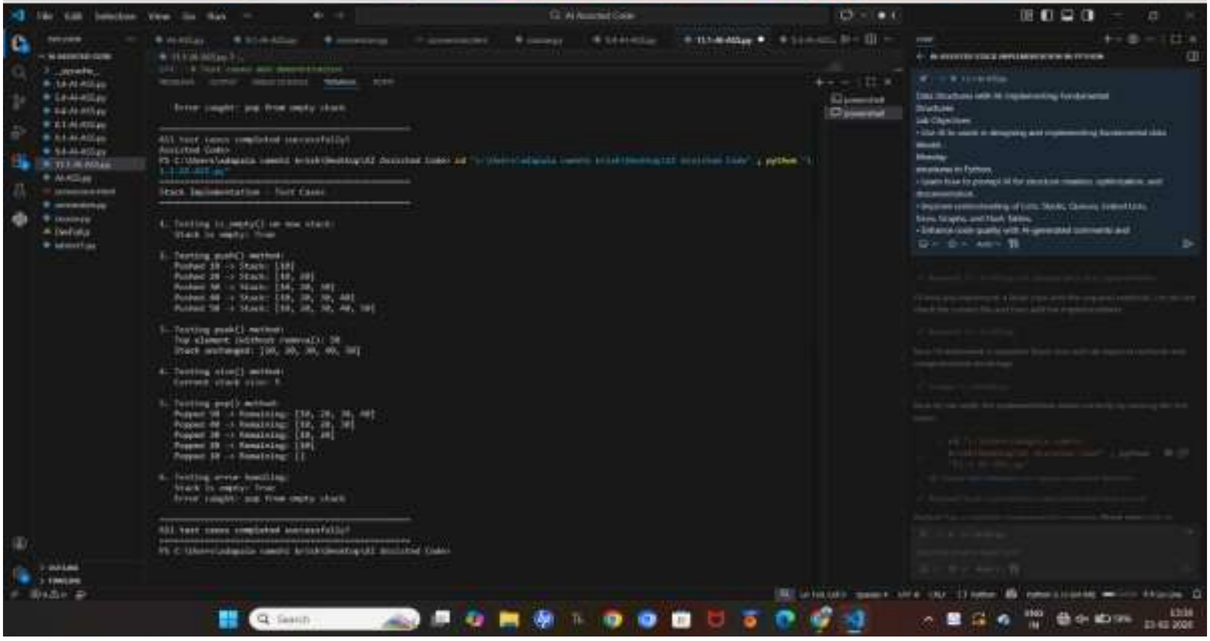
Code:











Observation:

The AI successfully generated a functional Stack implementation with all required methods and proper documentation. It handled edge cases using exceptions and used efficient list operations with O(1) time complexity. This task demonstrated that AI can speed up development, improve code readability, and help in better understanding fundamental data structures.

TASK-2

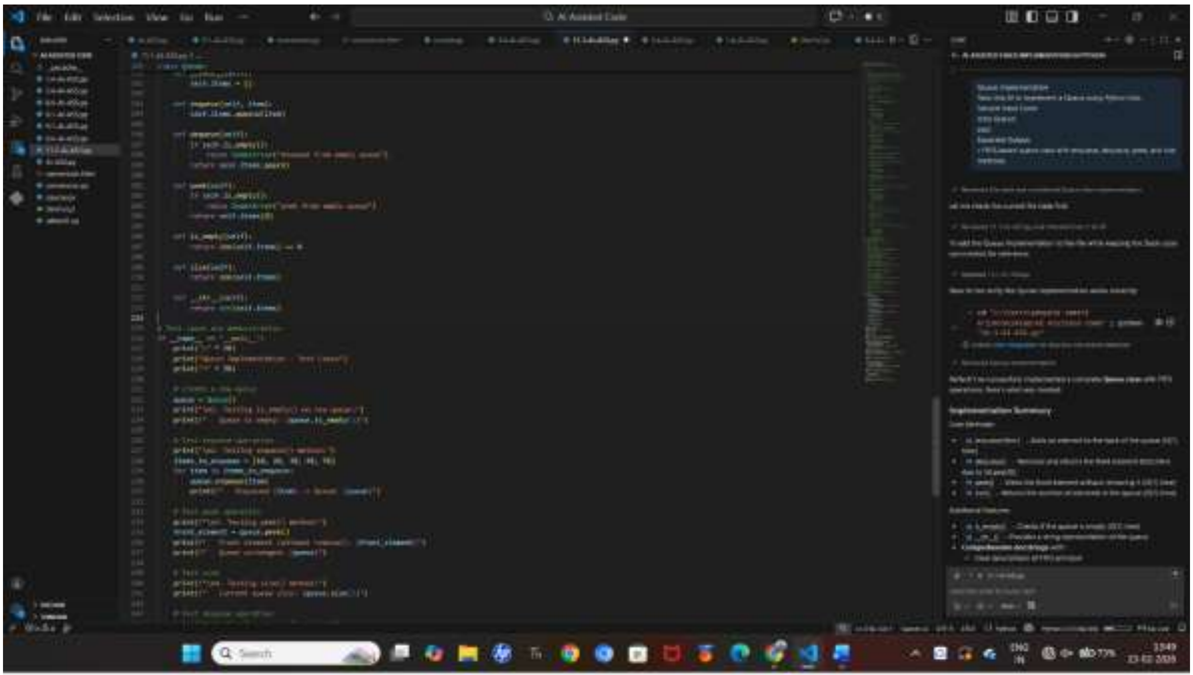
Scenario:

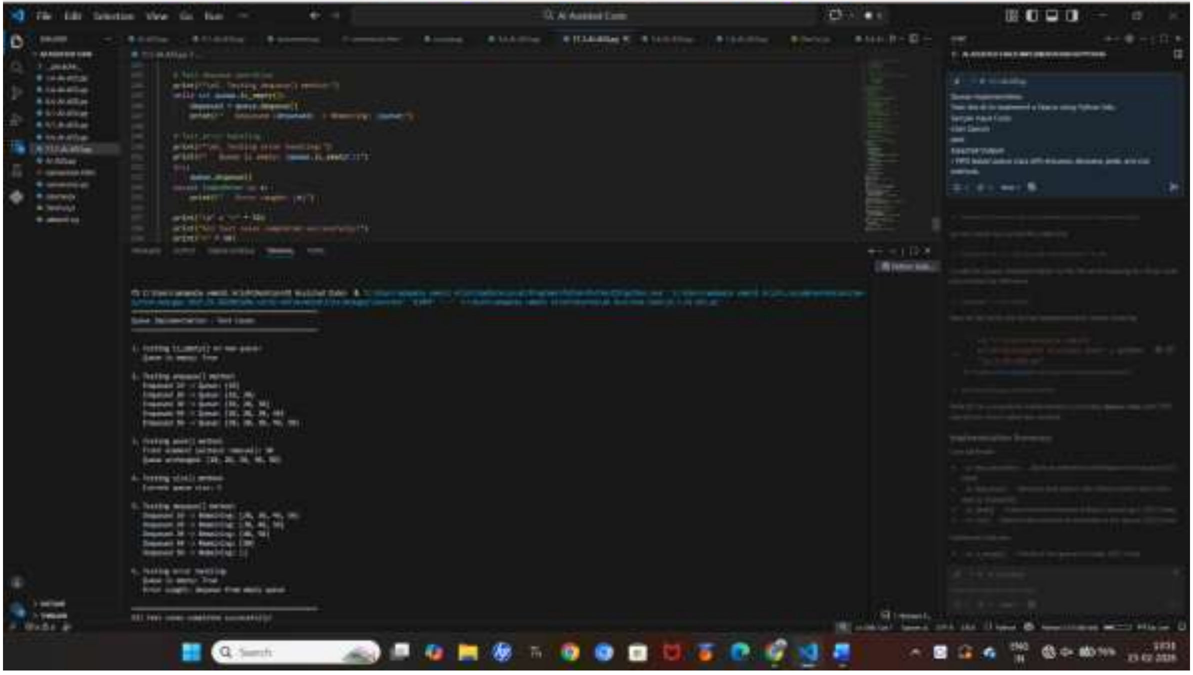
In this task, we used AI to implement a Queue data structure in Python using lists. The goal was to understand how a queue follows the FIFO (First In, First Out) principle and how AI can help in generating structured, clean, and efficient code. AI was prompted to create a functional Queue class with essential operations and proper handling of edge cases.

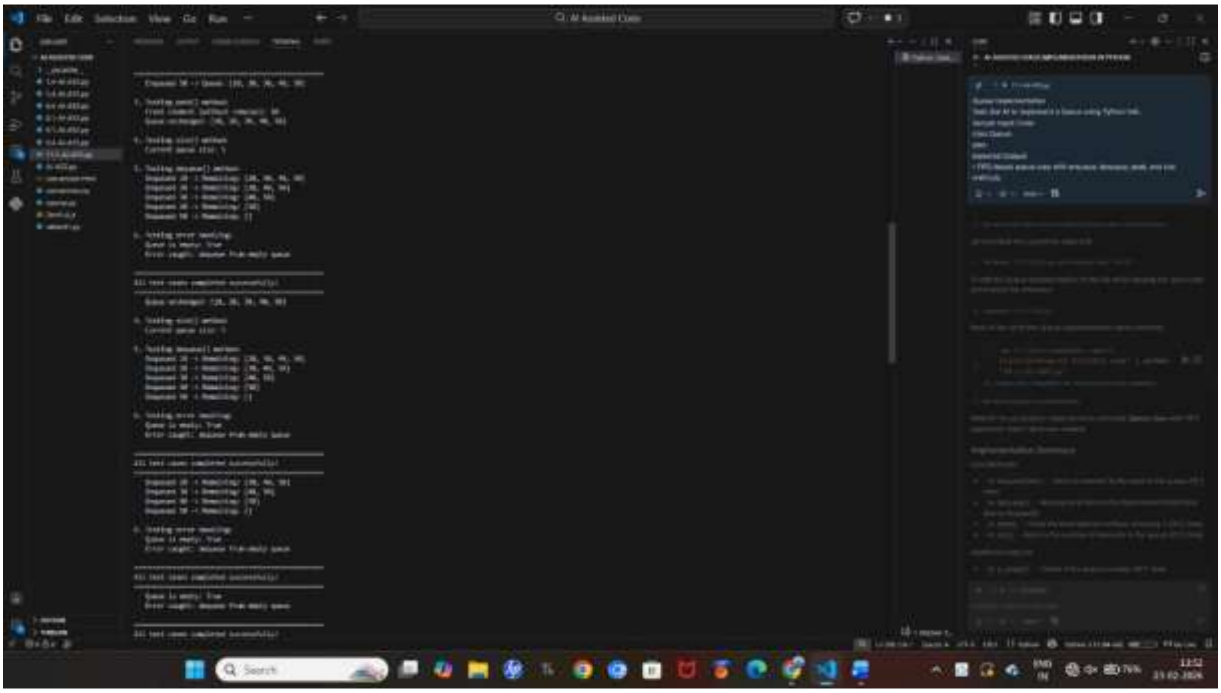
Prompt:

“Generate a beginner-friendly Python Queue class using a list. Include enqueue, dequeue, peek, and size methods. Follow FIFO principle and handle edge cases like dequeue from an empty queue. Add proper docstrings.”

Code:







Observation:

The AI generated a functional FIFO-based Queue class with all required methods. It properly handled empty queue conditions using exceptions and included documentation for clarity. The enqueue operation works efficiently, while dequeue from a list may take O(n) time due to element shifting. This task shows that AI can assist in writing clear, structured implementations of fundamental data structures and improve code quality.

TASK-3

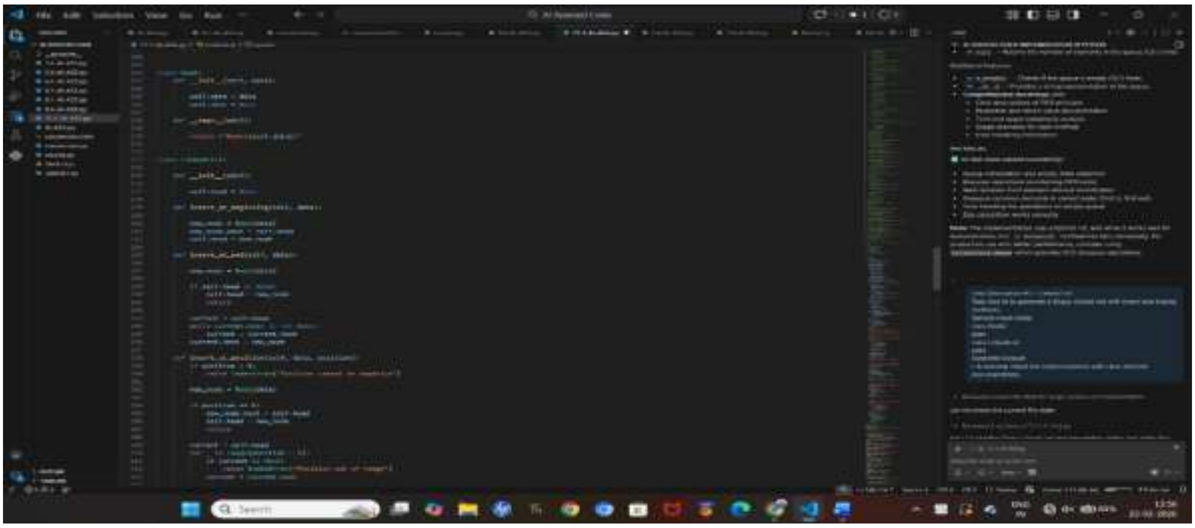
Scenario:

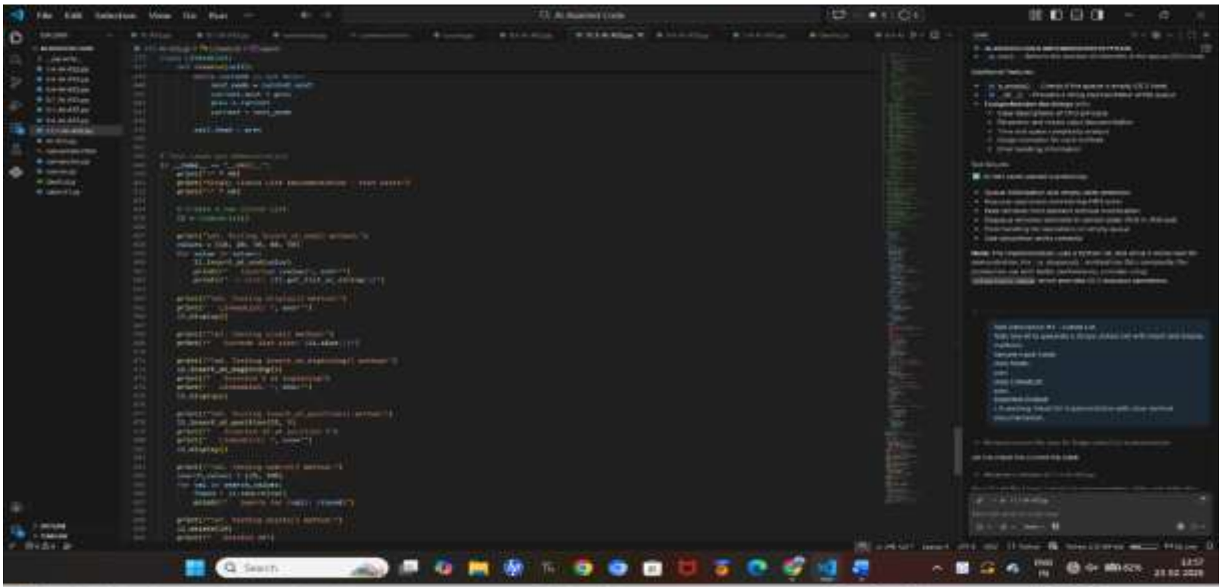
In this task, we used AI to generate a Singly Linked List implementation in Python. The objective was to understand how linked lists work using nodes and pointers instead of built-in lists. AI was used to create a Node class and a LinkedList class with insert and display methods, along with proper documentation for better understanding and readability.

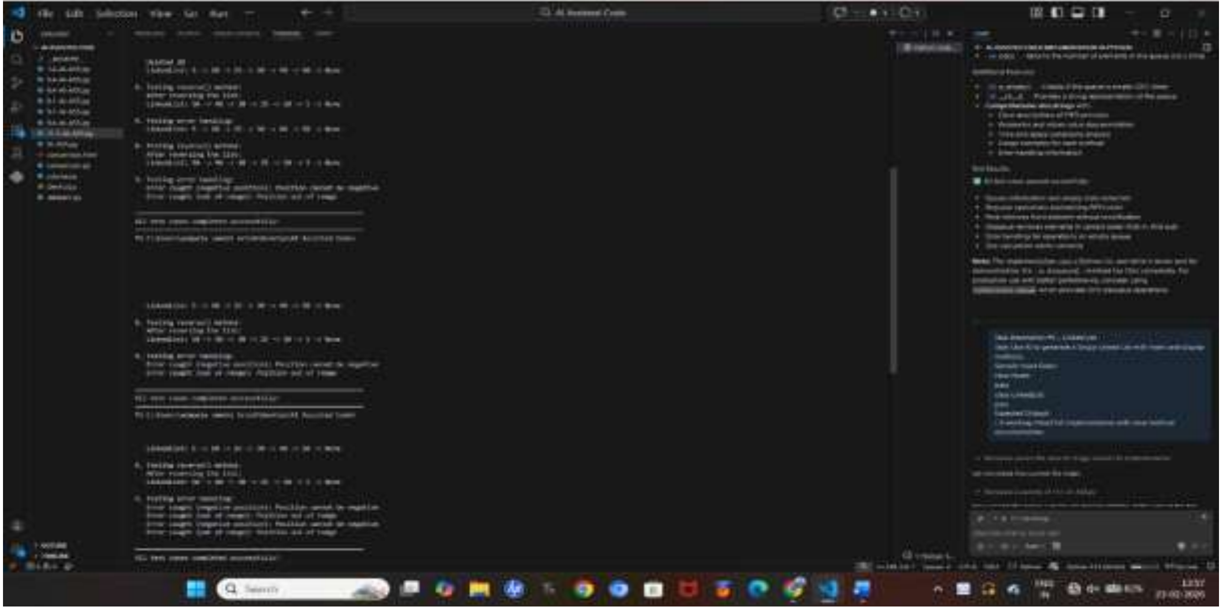
Prompt:

“Generate a beginner-friendly Python implementation of a Singly Linked List. Create a Node class and a LinkedList class. Include insert and display methods. Add clear docstrings and keep the implementation simple and well-structured.”

Code:







Observation:

The AI successfully generated a working Singly Linked List with proper class structure and method documentation. The insert method correctly adds new nodes, and the display method traverses and prints the list elements. The implementation clearly demonstrates how nodes are connected using references, improving understanding of dynamic memory structures. This task shows how AI can help simplify complex data structure implementations and improve code clarity.

TASK-4

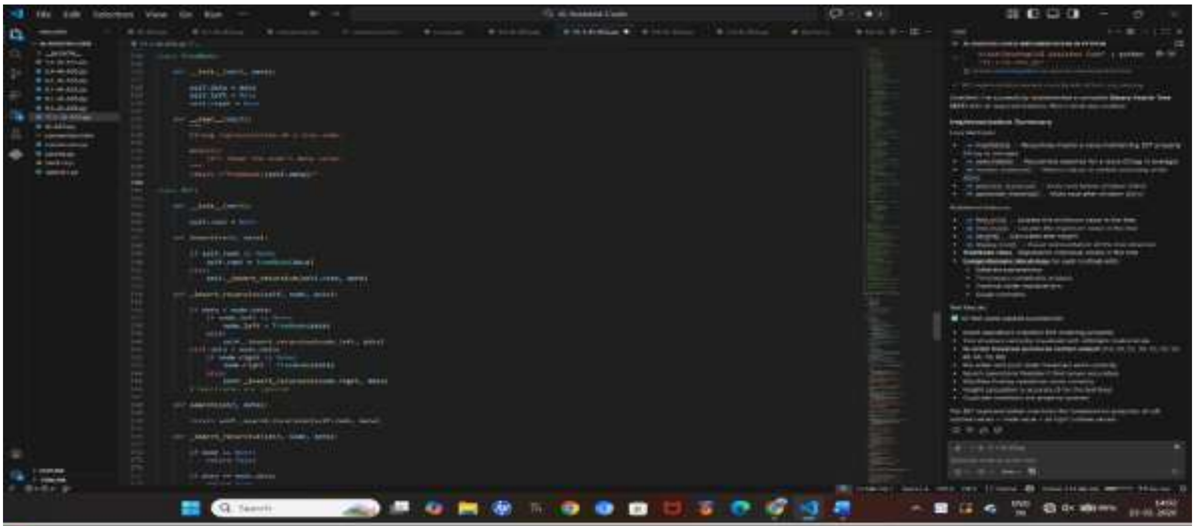
Scenario:

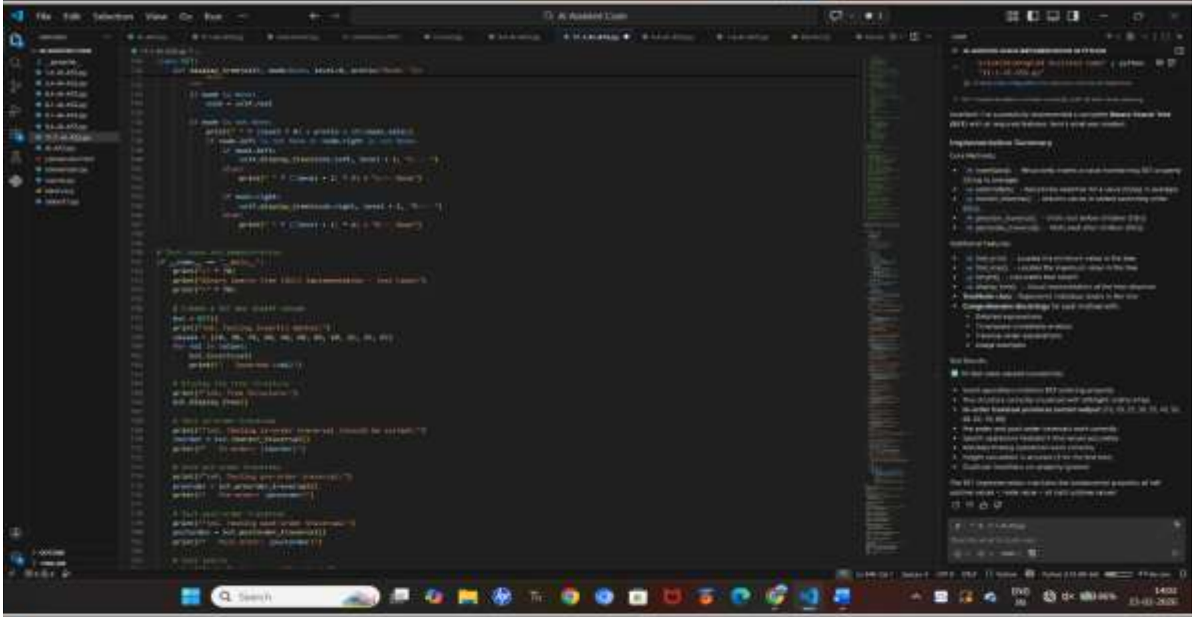
In this task, we used AI to implement a Binary Search Tree (BST) in Python. The objective was to understand how BST maintains sorted order using the property that left child values are smaller and right child values are larger than the root. AI was used to generate a BST class with recursive insert and in-order traversal methods to demonstrate how elements are stored and retrieved in sorted order.

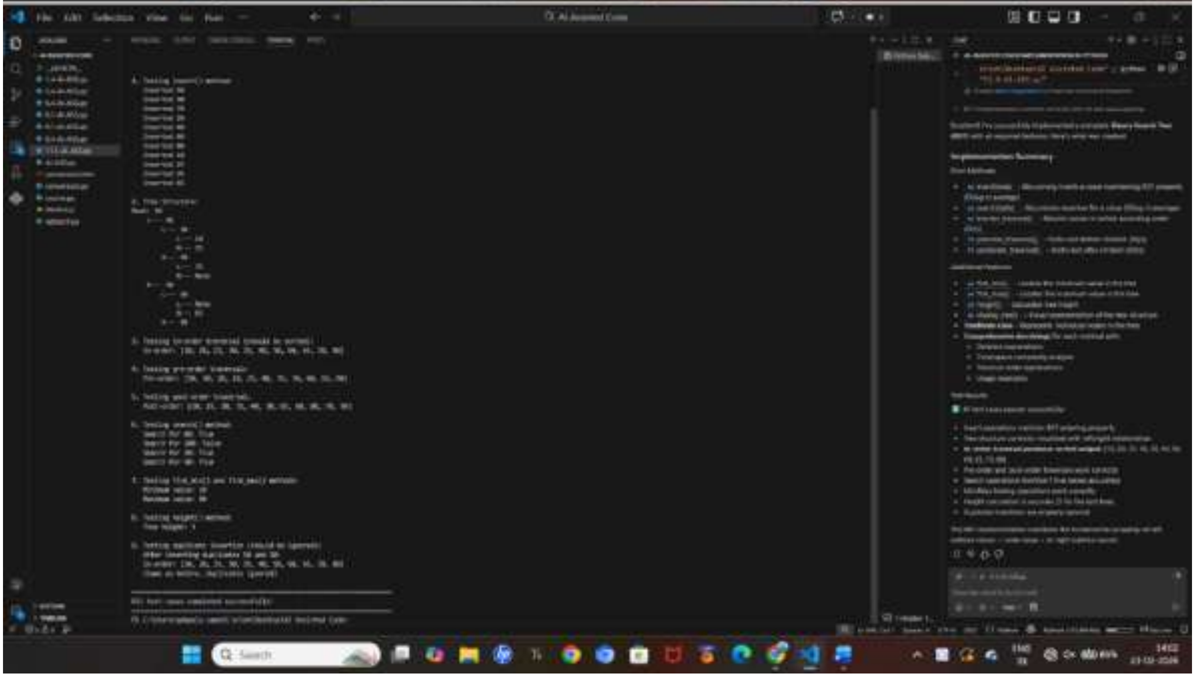
Prompt:

“Generate a beginner-friendly Python implementation of a Binary Search Tree (BST). Include recursive insert and in-order traversal methods. Keep the structure simple and add proper docstrings for clarity.”

Code:







Observation:

The AI successfully generated a BST implementation with recursive insert and in-order traversal methods. The insert method correctly places nodes according to BST rules, and the in-order traversal prints elements in sorted order. The recursive approach improves understanding of tree traversal logic. This task demonstrates that AI can effectively assist in implementing hierarchical data structures with clear and well-documented code.

TASK-5

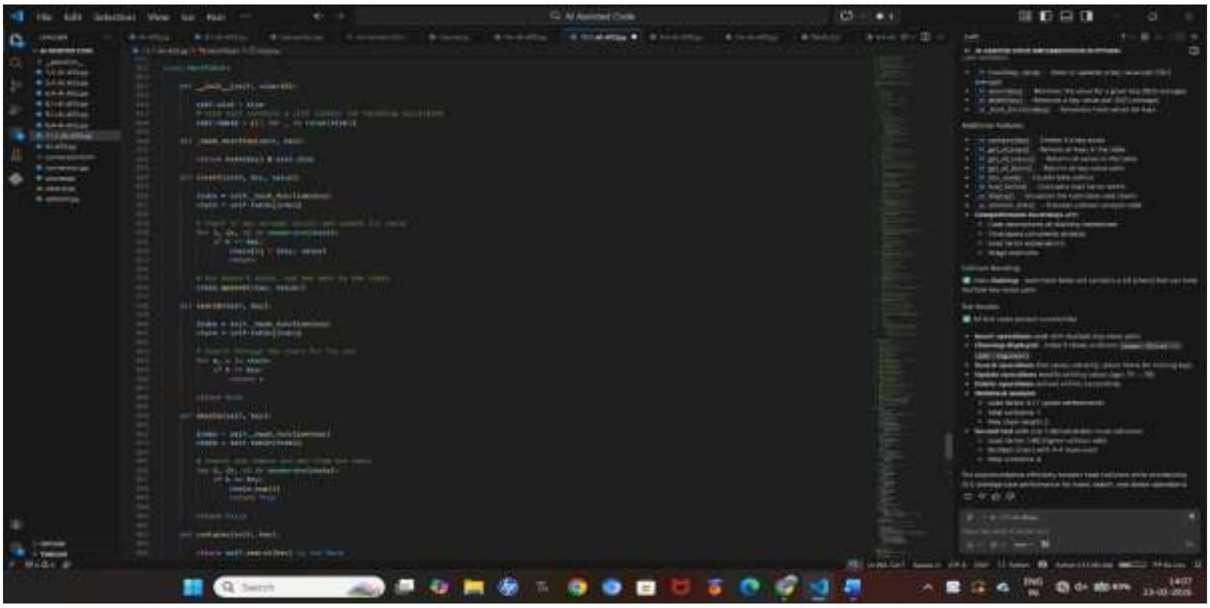
Scenario:

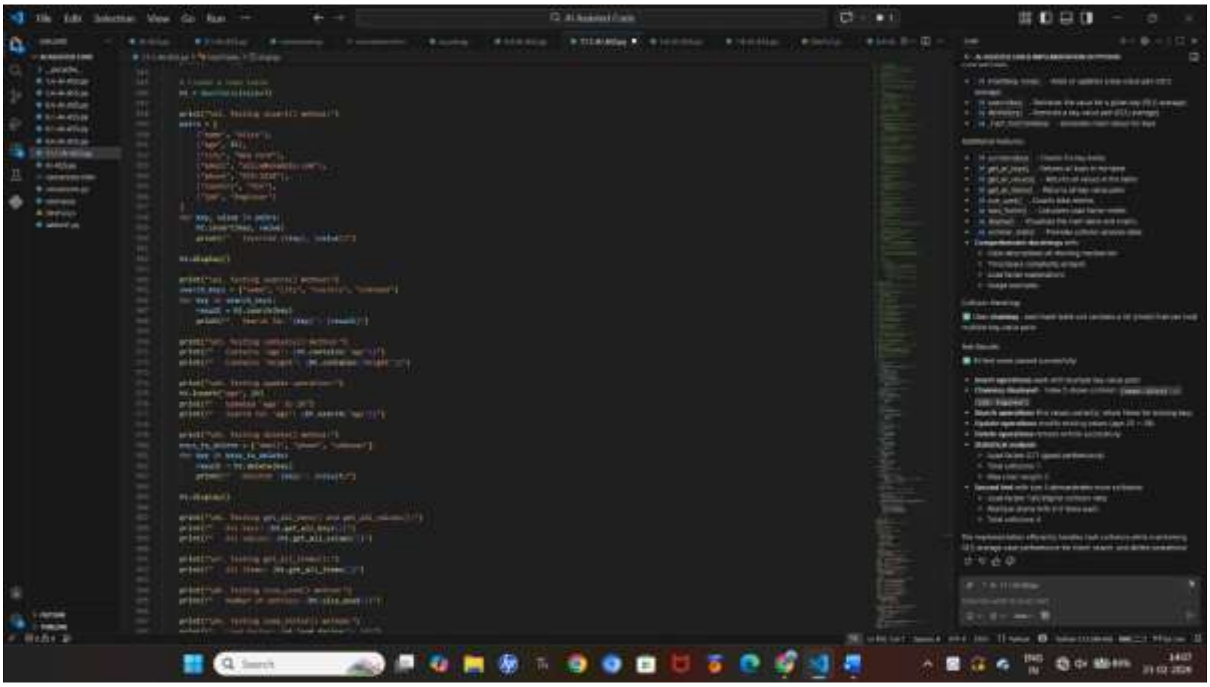
In this task, we used AI to implement a Hash Table in Python with basic operations like insert, search, and delete. The objective was to understand how hashing works for fast data access and how collisions are handled using chaining (storing multiple elements in the same bucket). AI was used to generate a structured and well-documented implementation to clearly demonstrate these concepts.

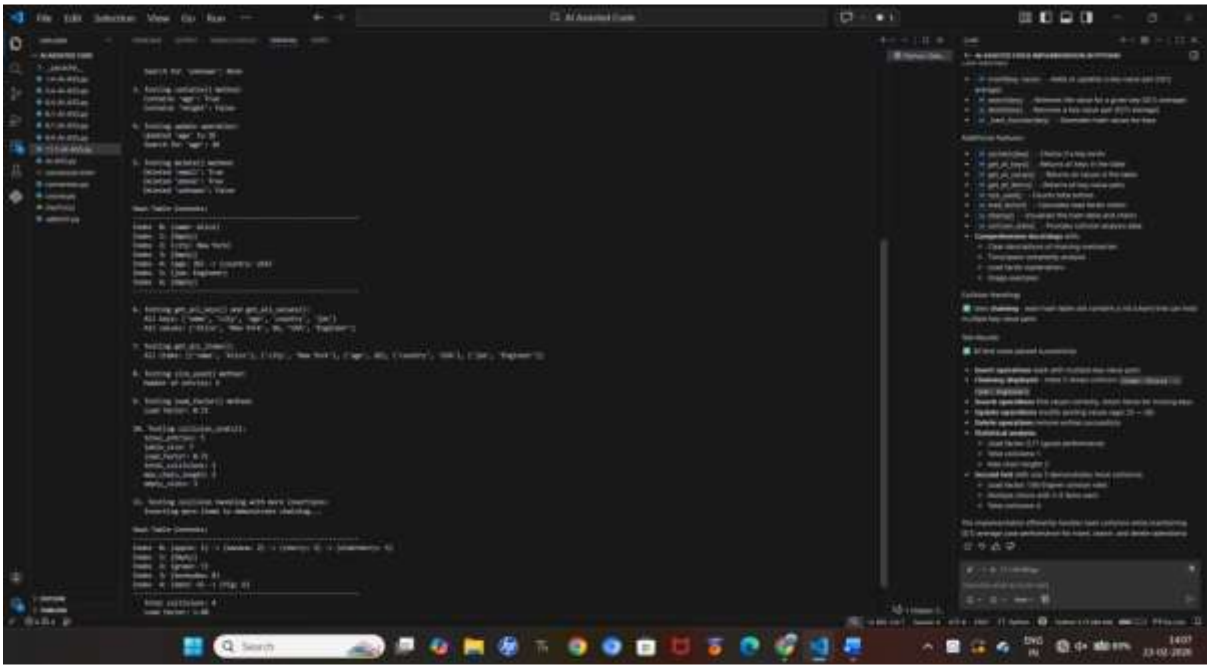
Prompt:

“Generate a beginner-friendly Python HashTable class. Include insert, search, and delete methods. Implement collision handling using chaining (lists inside buckets). Add clear comments and keep the implementation simple.”

Code:







Observation:

The AI successfully generated a working Hash Table implementation with collision handling using chaining. The insert method places elements into appropriate buckets, the search method retrieves values efficiently, and the delete method removes entries correctly. The code was well-commented and easy to understand. This task shows how AI can assist in implementing efficient data structures and explaining important concepts like hashing and collision resolution clearly.

TASK-6

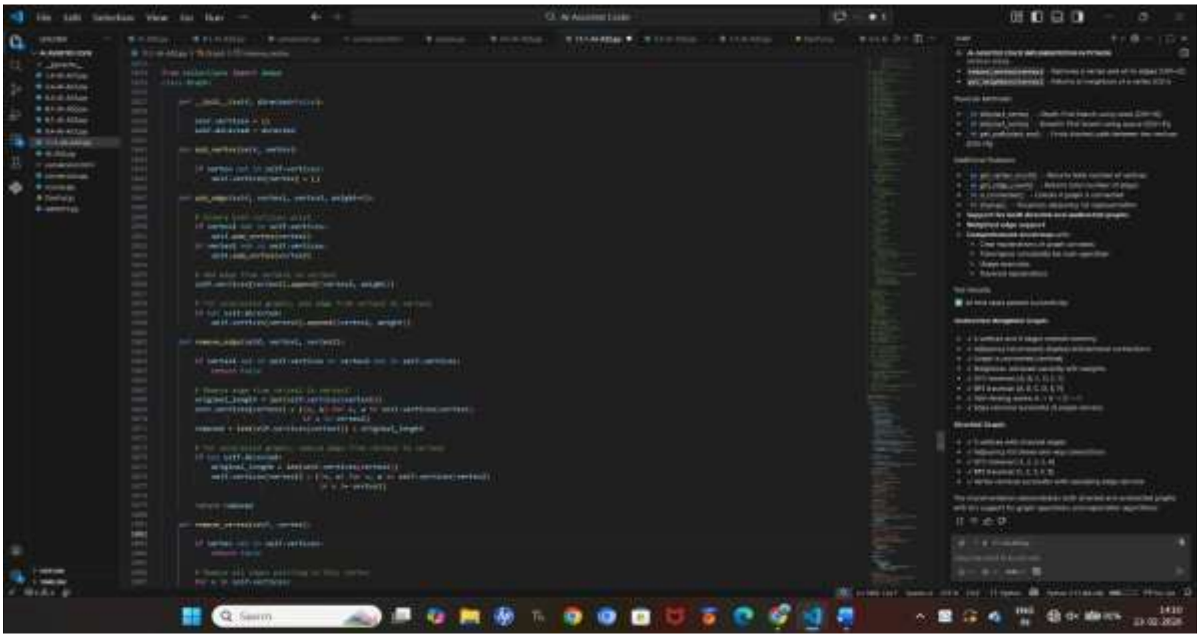
Scenario:

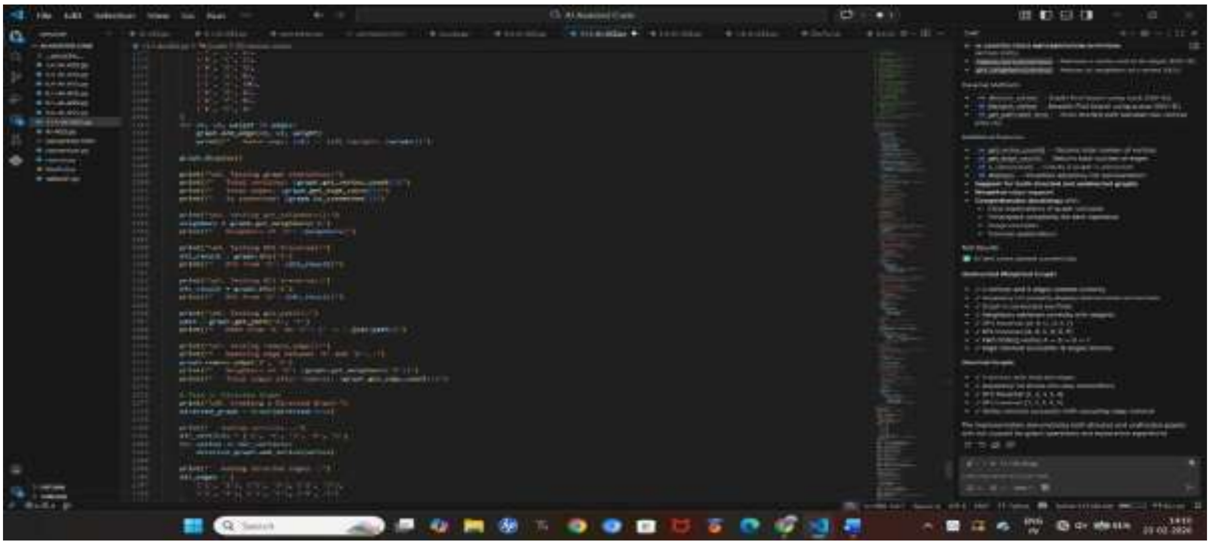
In this task, we used AI to implement a Graph data structure using an adjacency list representation in Python. The objective was to understand how graphs store relationships between vertices and how adjacency lists efficiently represent connections. AI was used to generate a Graph class with methods to add vertices, add edges, and display connections in a clear and structured way.

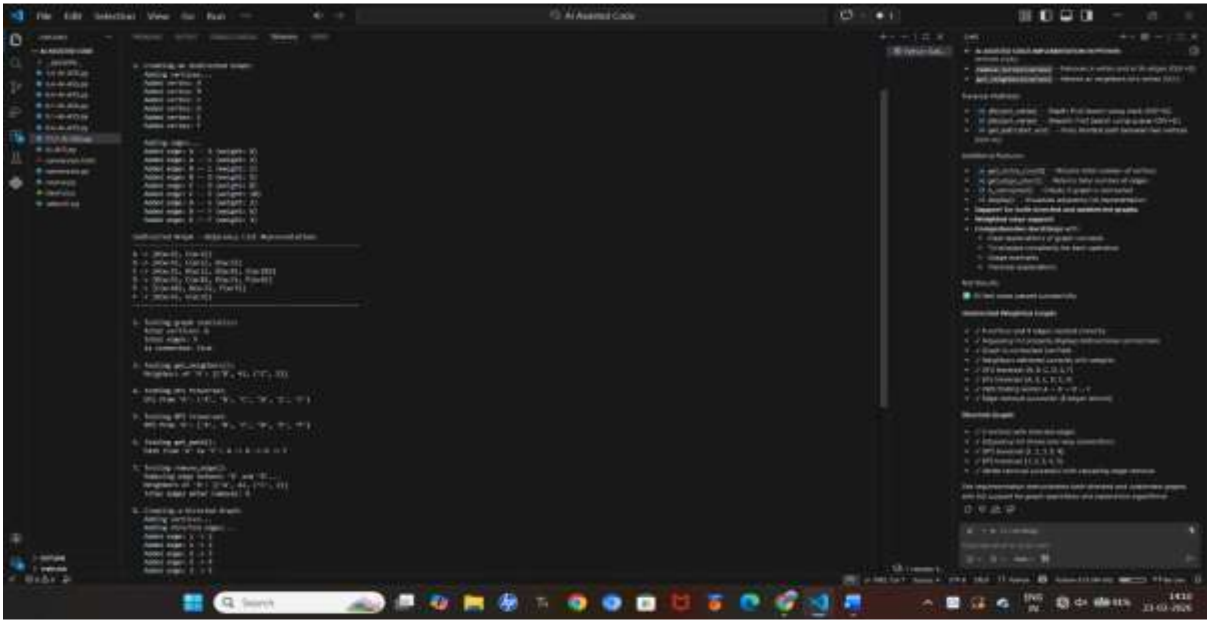
Prompt:

“Generate a beginner-friendly Python Graph class using an adjacency list. Include methods to add vertices, add edges, and display connections. Keep the implementation simple and add proper comments for clarity.”

Code:







Observation:

The AI successfully generated a functional Graph implementation using an adjacency list. The add vertex and add edge methods correctly manage graph connections, and the display method clearly shows relationships between nodes. The structure is efficient for sparse graphs and easy to understand. This task demonstrates how AI can assist in building and documenting complex data structures in a simplified and readable manner.”

TASK-7

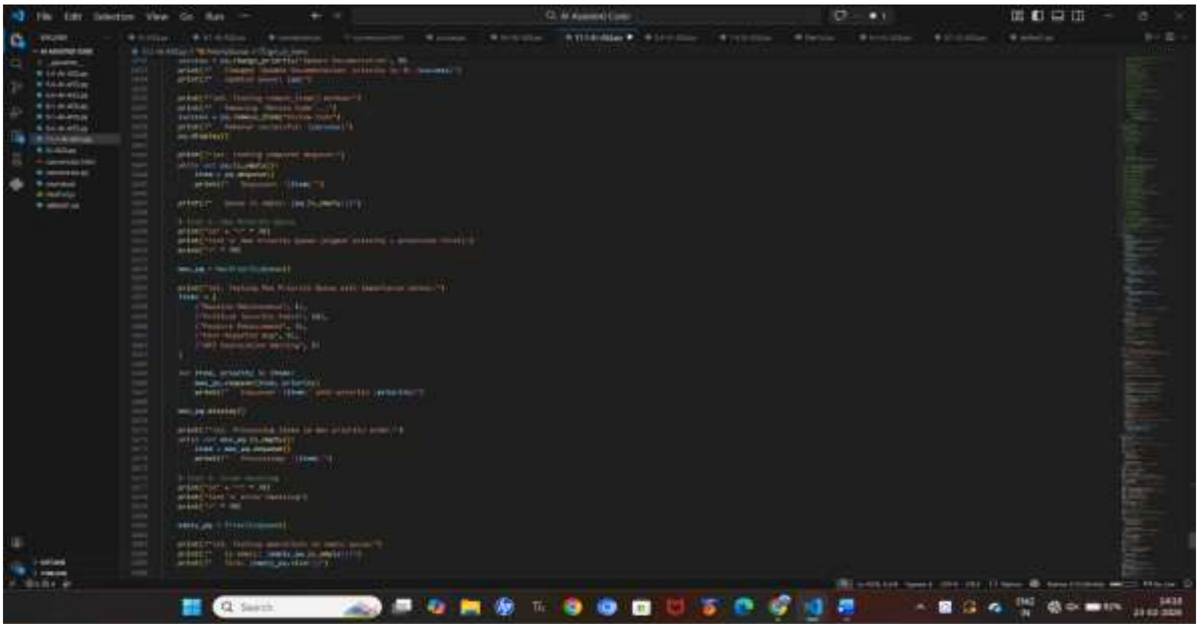
Scenario:

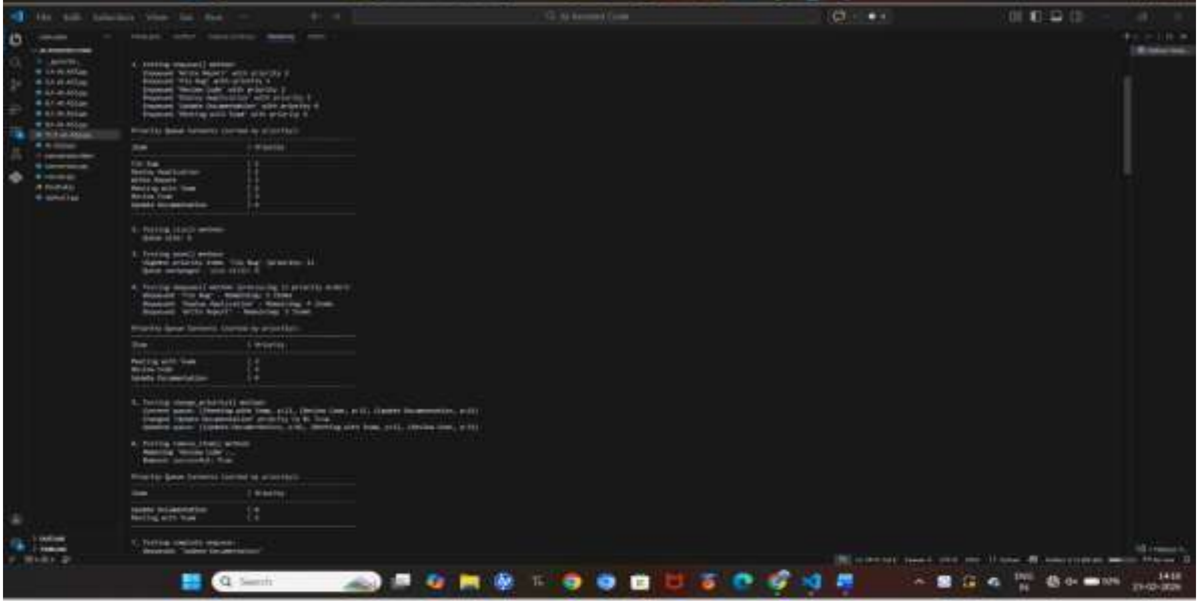
In this task, we used AI to implement a Priority Queue in Python using the built-in heapq module. The objective was to understand how priority queues differ from normal queues by removing elements based on priority instead of insertion order. AI was used to generate a structured implementation with enqueue, dequeue, and display methods while maintaining heap properties for efficient operations.

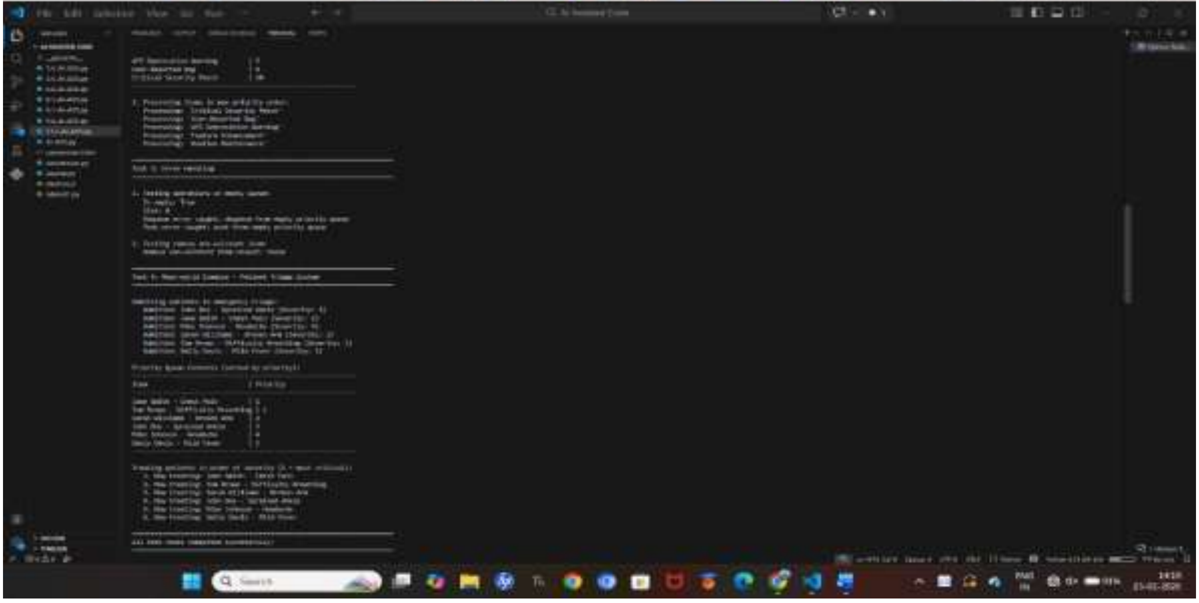
Prompt:

“Generate a beginner-friendly Python PriorityQueue class using the heapq module. Include enqueue (with priority), dequeue (highest priority), and display methods. Keep the implementation simple and add clear comments.”

Code:







Observation:

The AI successfully generated a working Priority Queue implementation using heapq, which maintains the heap property for efficient insertion and removal. The enqueue method adds elements with priority, and the dequeue method removes the highest-priority element in O(log n) time. The implementation was clean, well-commented, and efficient. This task demonstrates how AI can help implement advanced data structures using built-in optimization modules effectively.”

TASK-8

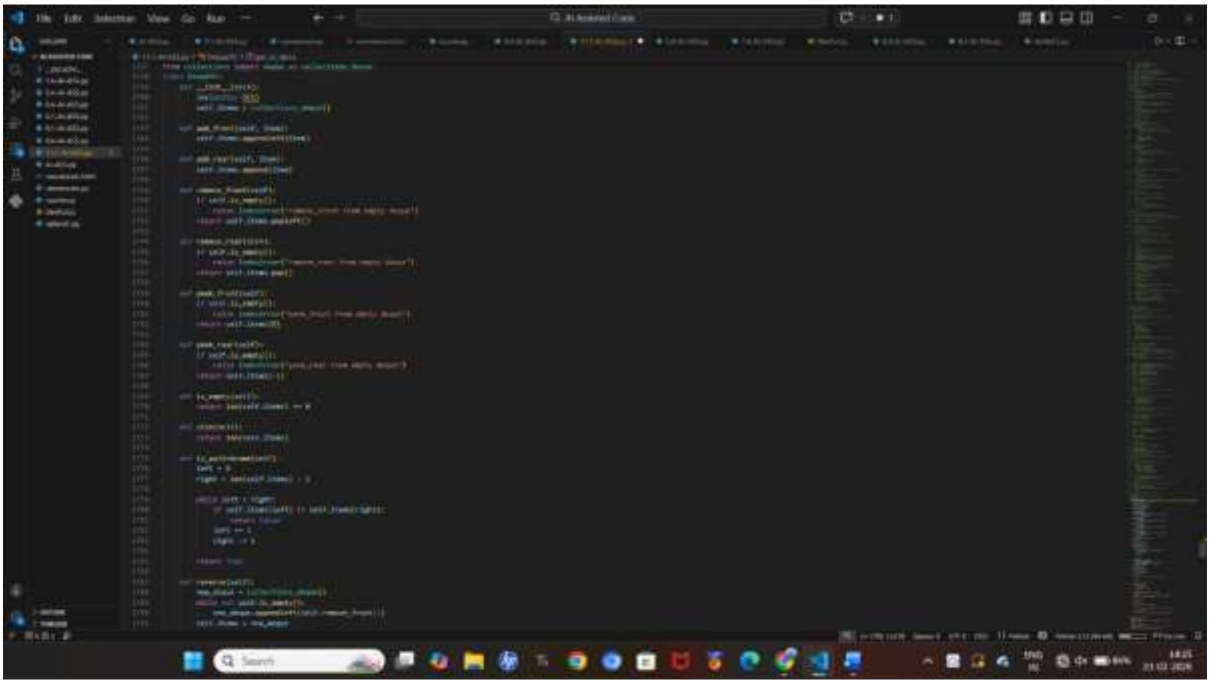
Scenario:

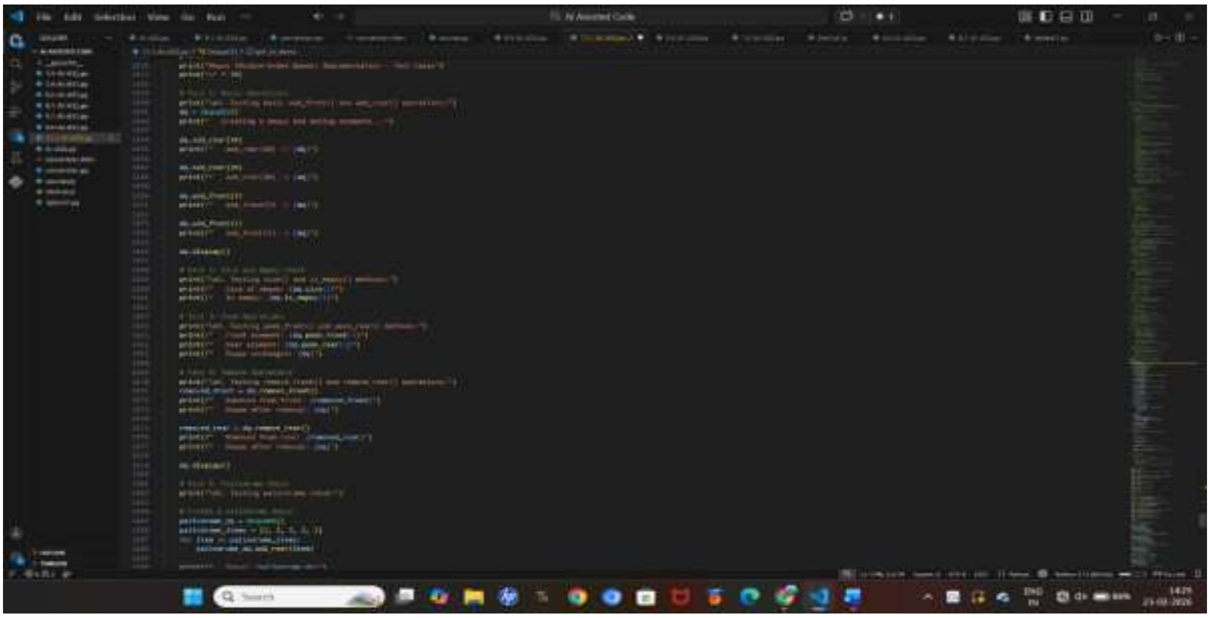
In this task, we used AI to implement a Deque (Double-Ended Queue) in Python using the collections.deque module. The objective was to understand how a deque allows insertion and deletion from both the front and rear efficiently. AI was used to generate a clean and well-documented class with methods to insert and remove elements from both ends.

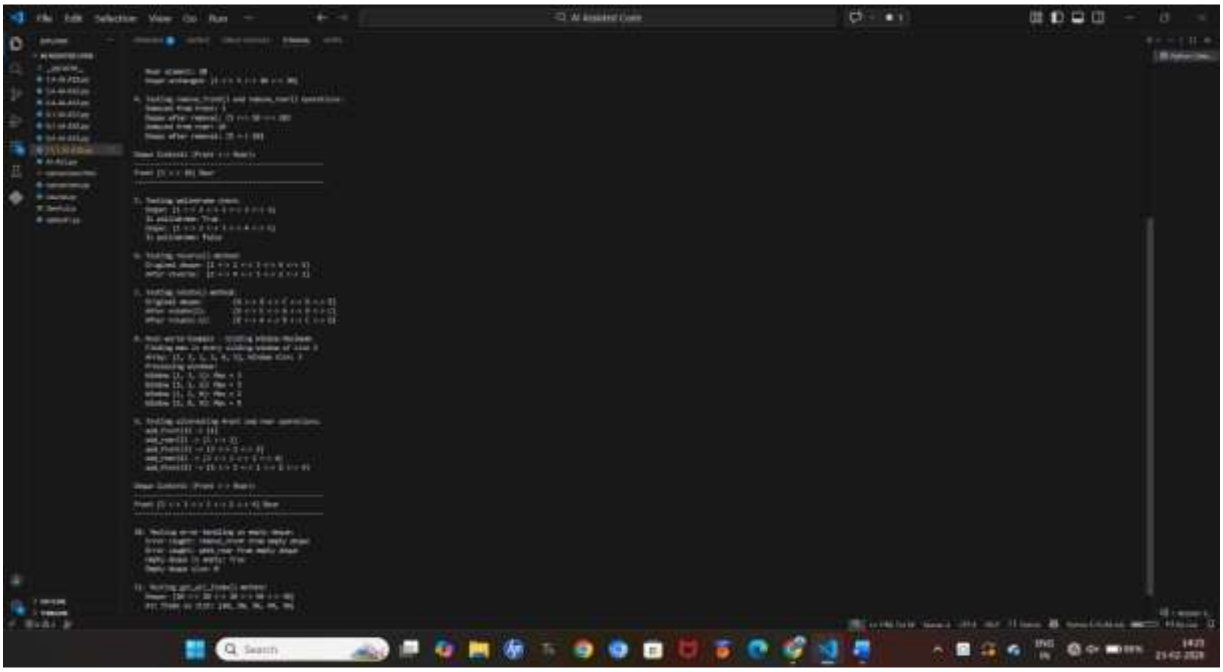
Prompt:

“Generate a beginner-friendly Python class for a Deque using collections.deque. Include methods to insert and remove elements from both front and rear. Add proper docstrings and keep the implementation simple.”

Code:







Observation:

The AI successfully generated a functional Deque implementation using collections. deque, which supports O(1) time complexity for insertions and deletions at both ends. The methods were clearly structured and properly documented. This task demonstrates how AI can help implement efficient double-ended data structures while improving code readability and understanding.”

TASK-9

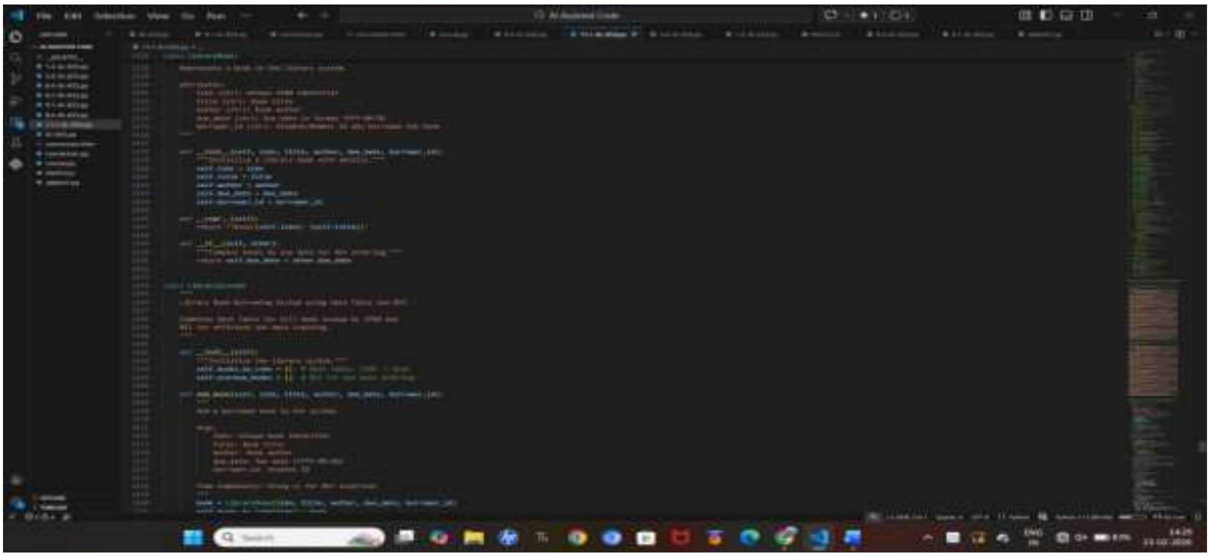
Scenario:

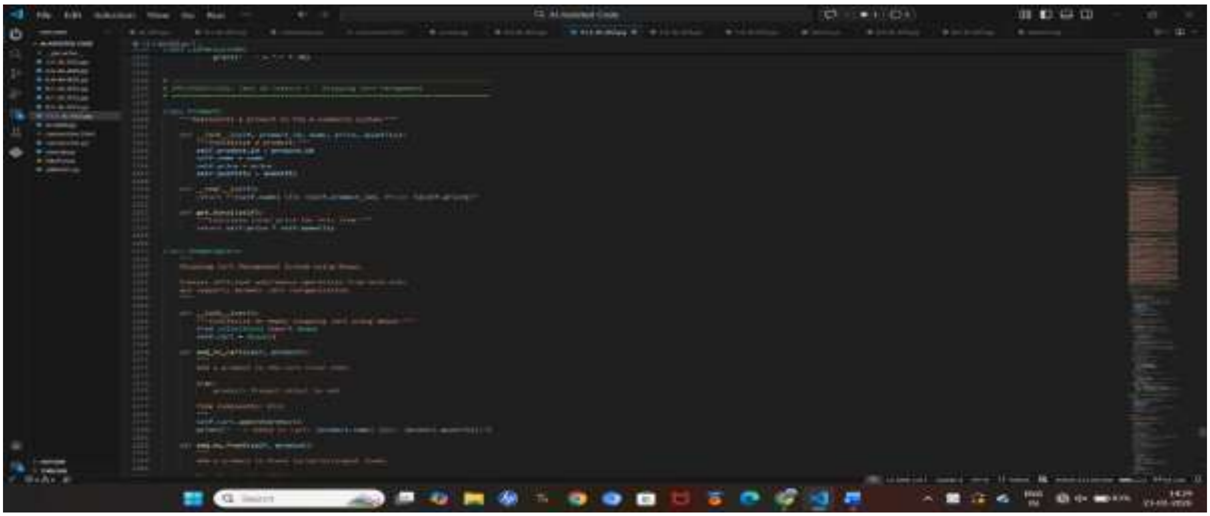
In this task, we analyzed different modules of a Campus Resource Management System and selected appropriate data structures for each feature based on functionality and efficiency. The goal was to match real-world problems like attendance tracking, event registration, library management, bus scheduling, and cafeteria service with the most suitable data structures. One feature was implemented using AI-assisted code generation to demonstrate practical application.

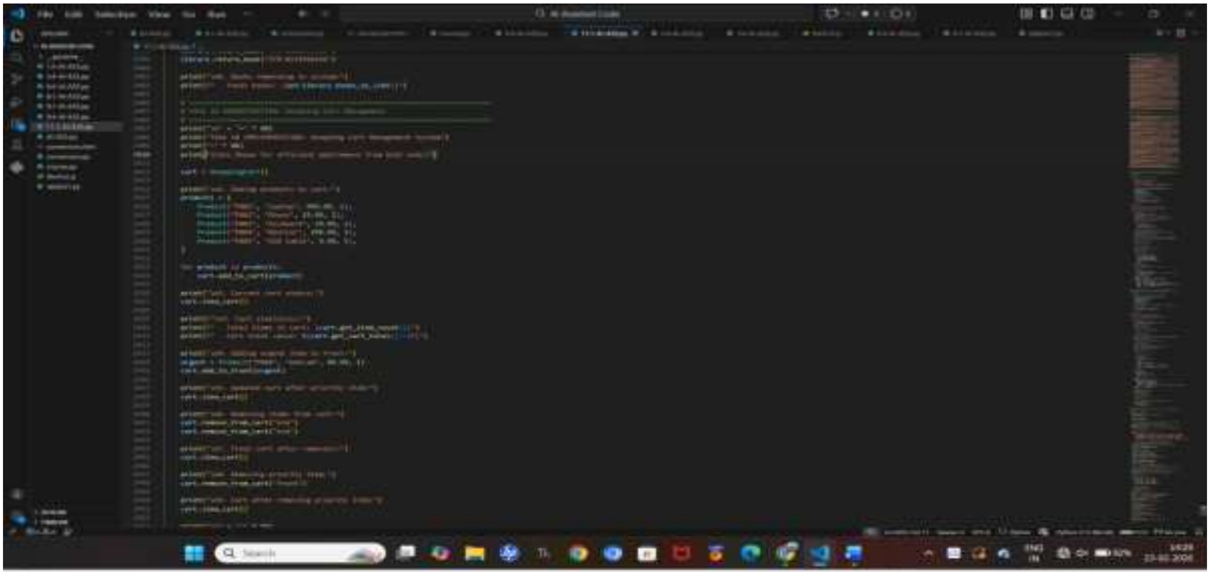
Prompt:

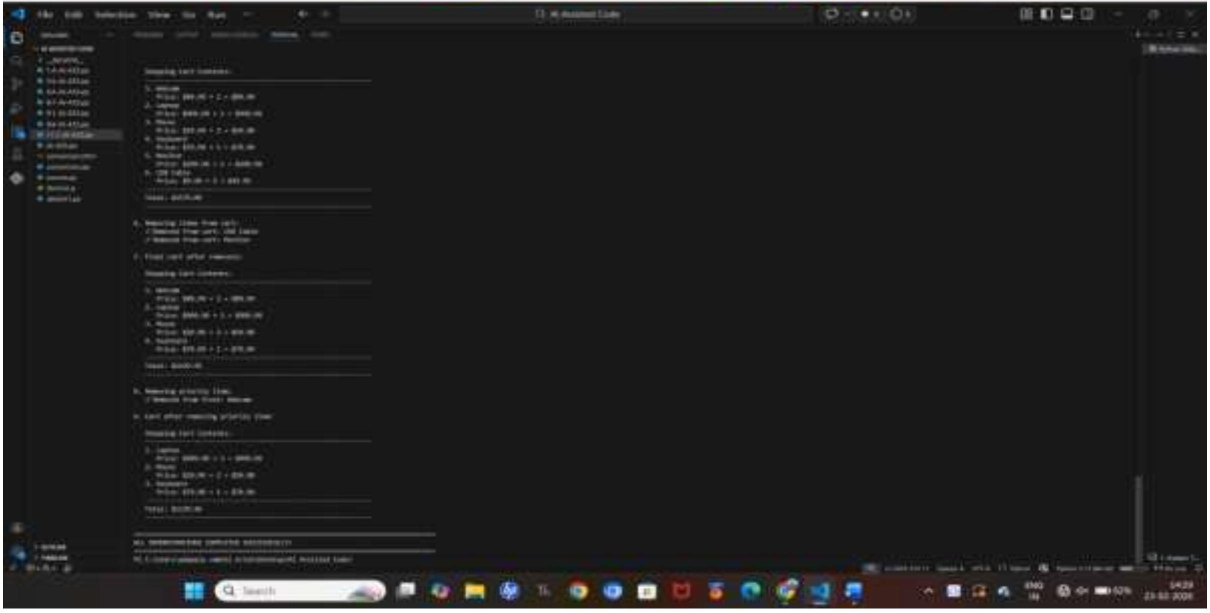
“Evaluate the Smart E-Commerce Platform features and choose the most appropriate data structure for each from Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, and Deque. Provide 2–3 sentence justifications for each choice. Generate a well-commented Python implementation for one selected feature.”

Code:









Observation:

The AI successfully selected optimal data structures such as Queue for order processing, Priority Queue for ranking products, Hash Table for fast product lookup, and Graph for delivery routing. The implementation was efficient and clearly documented. This activity showed how understanding data structure behavior helps in designing scalable systems and how AI enhances productivity and code clarity.