# ASSIGNMENT-11.1

NAME: T.Sai Tanuj

HALLTICKET NUNMER: 2403A52413

BATCH:15

## TASK-1

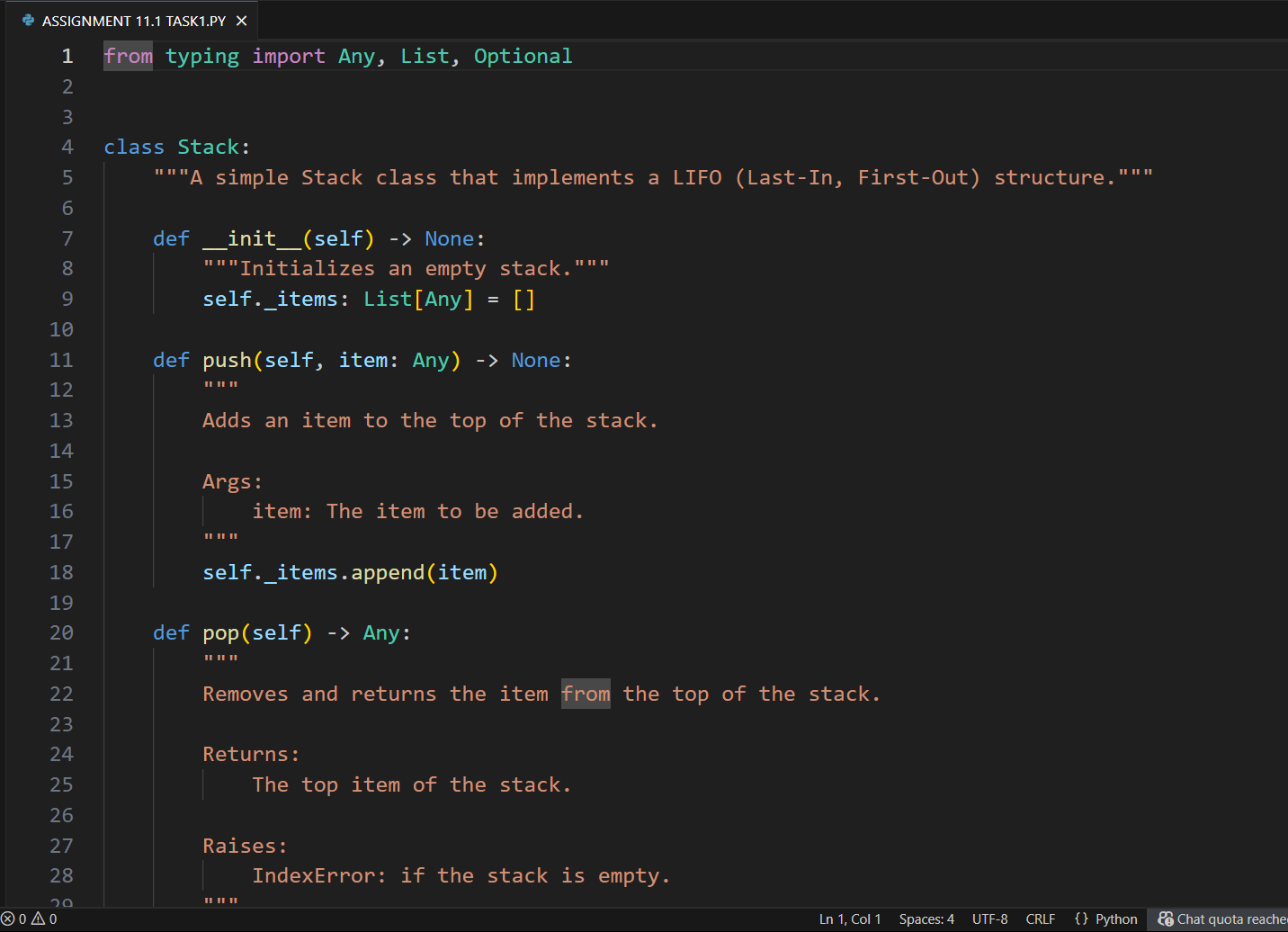
**QUESTION**

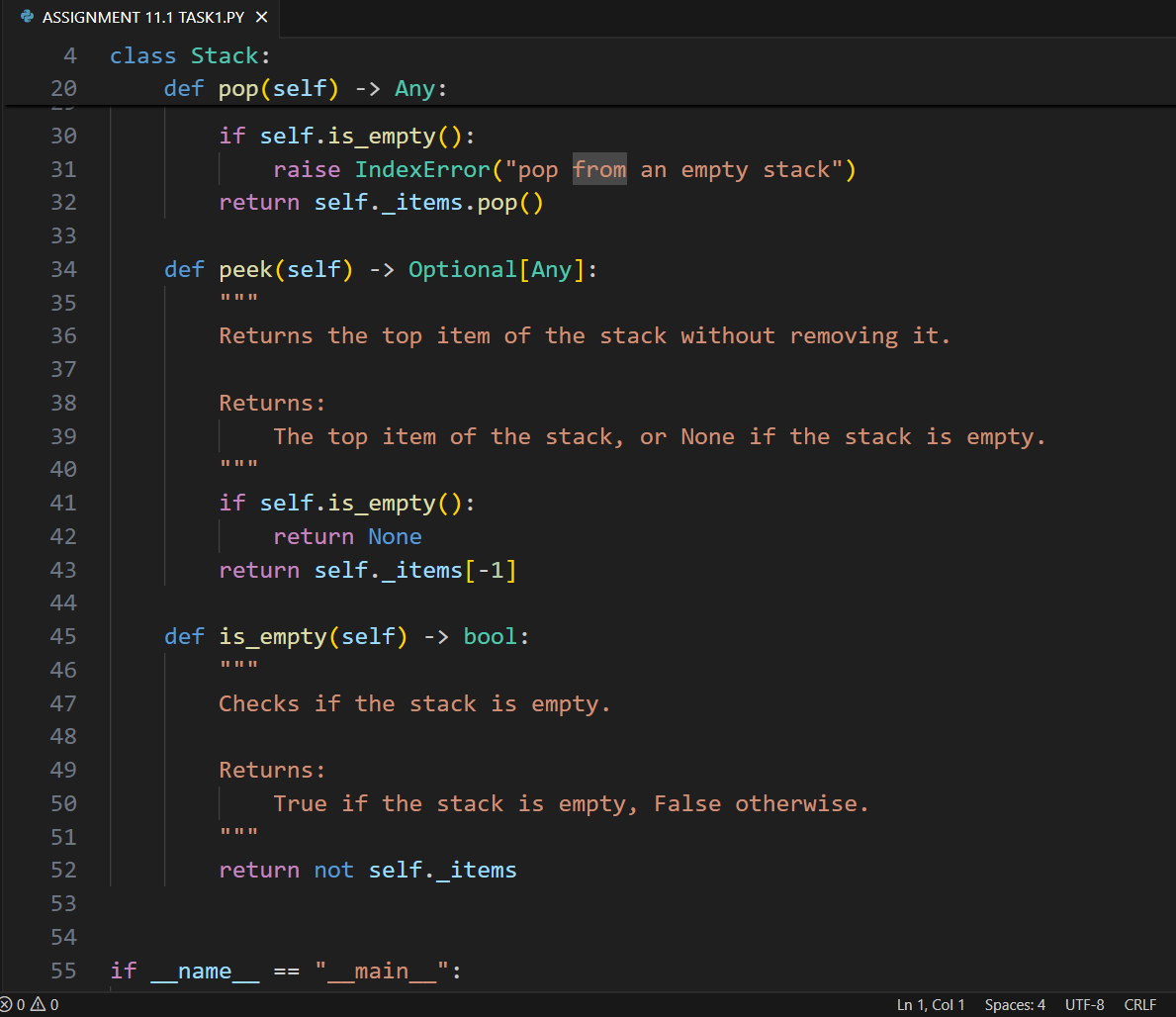
**#1 – Stack Implementation  
Task: Use AI to generate a Stack class with push, pop, peek, and is\_empty  
methods.  
Sample Input Code:  
class Stack:  
pass  
Expected Output:  
• A functional stack implementation with all required methods and  
docstrings**

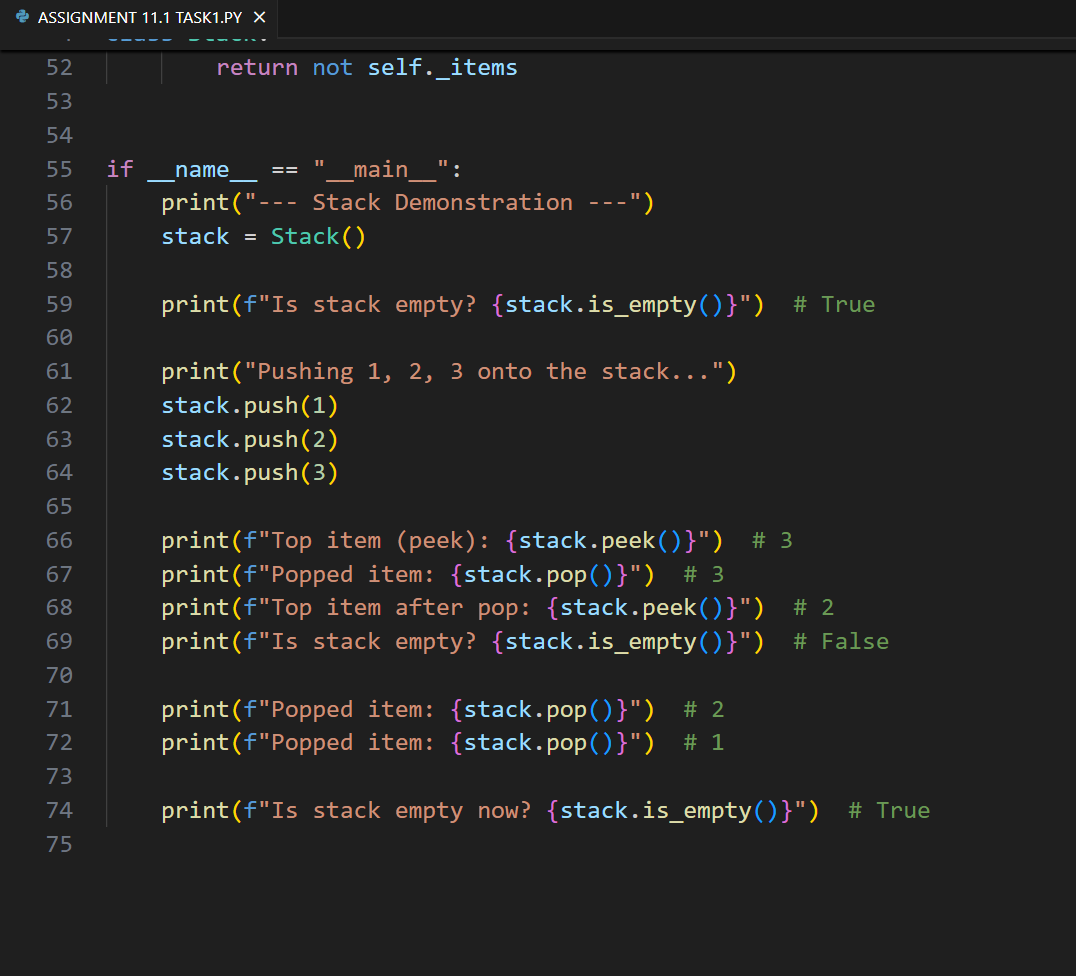
**PROMPT:**

**Write a Python class called Stack with methods push, pop, peek, and is empty. Use a list to store elements. Add short docstrings for the class and methods.**

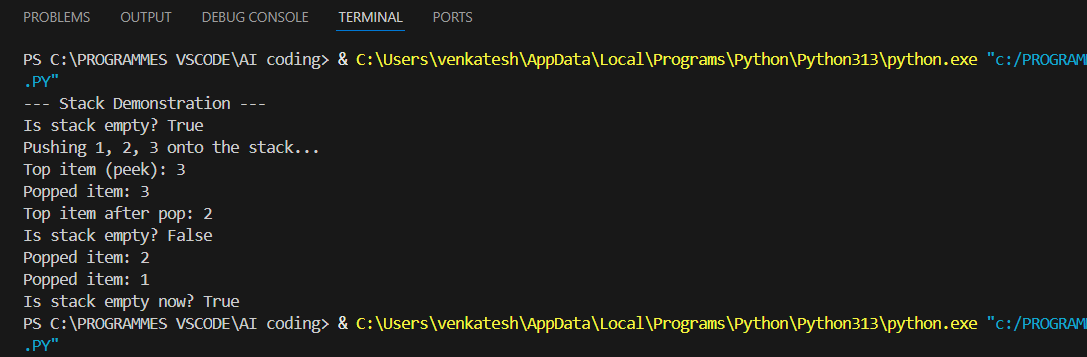
**CODE:**

****

****

****

**OUTPUT:**

****

**OBSERVATION:**

* **The Stack class correctly implements a LIFO structure using Python lists.**
* **It supports essential operations: push, pop, peek, and is\_empty with proper error handling**
* **Docstrings make the code easy to read and understand**
* **The demo shows stack behavior clearly, proving the implementation works as expected.**

## TASK-2

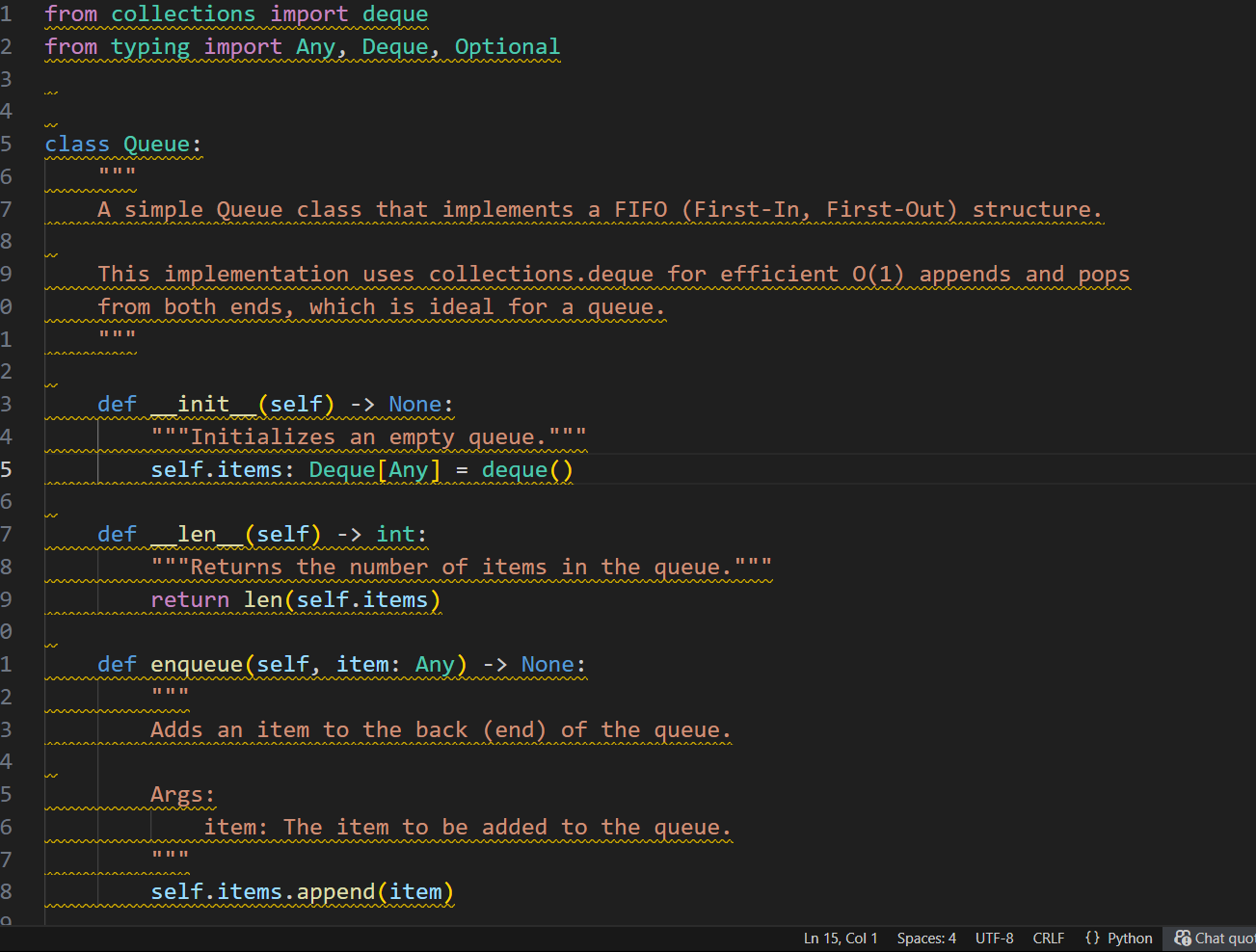
**QUESTION:**

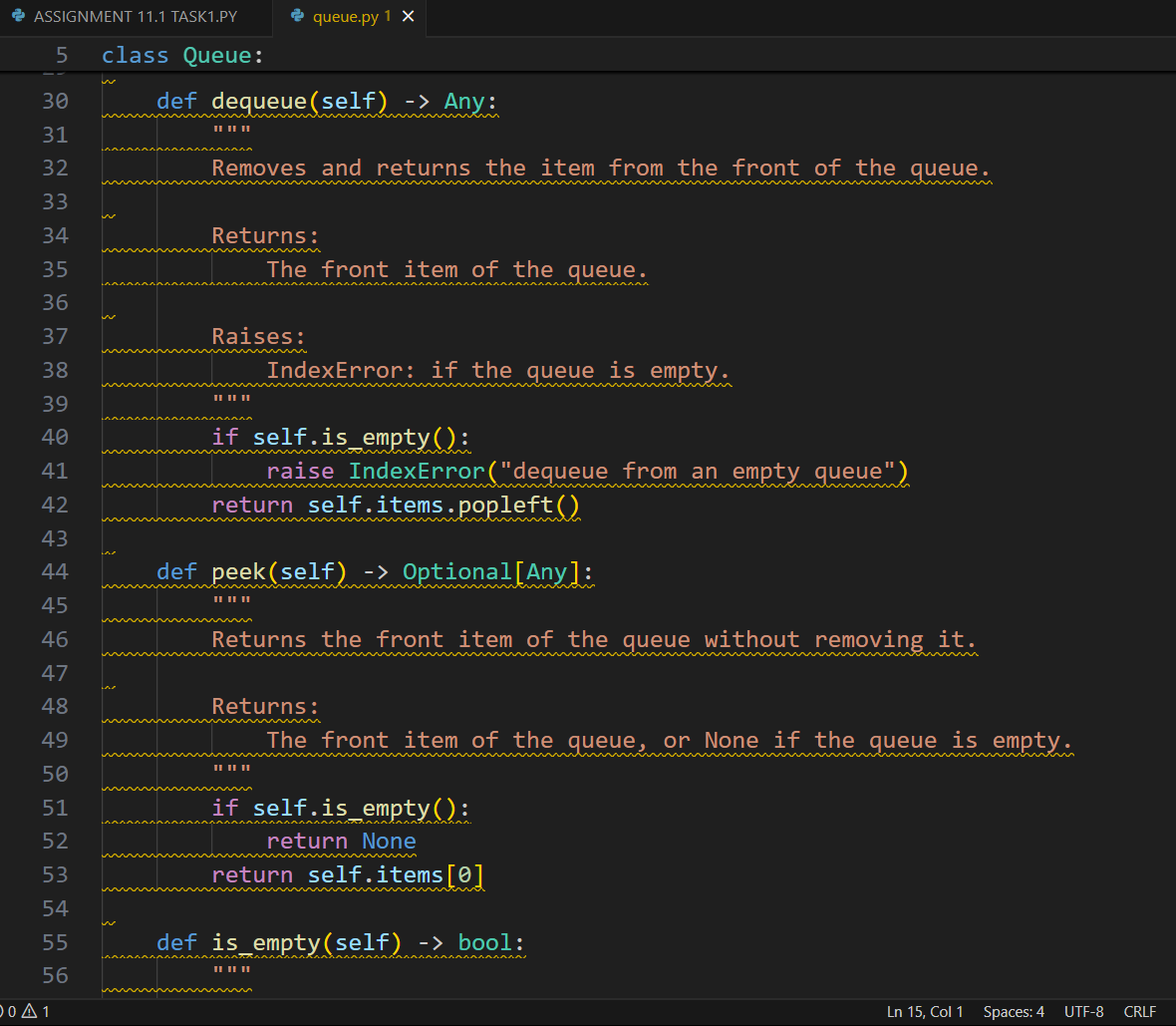
**2 – Queue Implementation  
Task: Use AI to implement a Queue using Python lists.  
Sample Input Code:  
class Queue:  
pass  
Expected Output:  
• FIFO-based queue class with enqueue, dequeue, peek, and size  
method.**

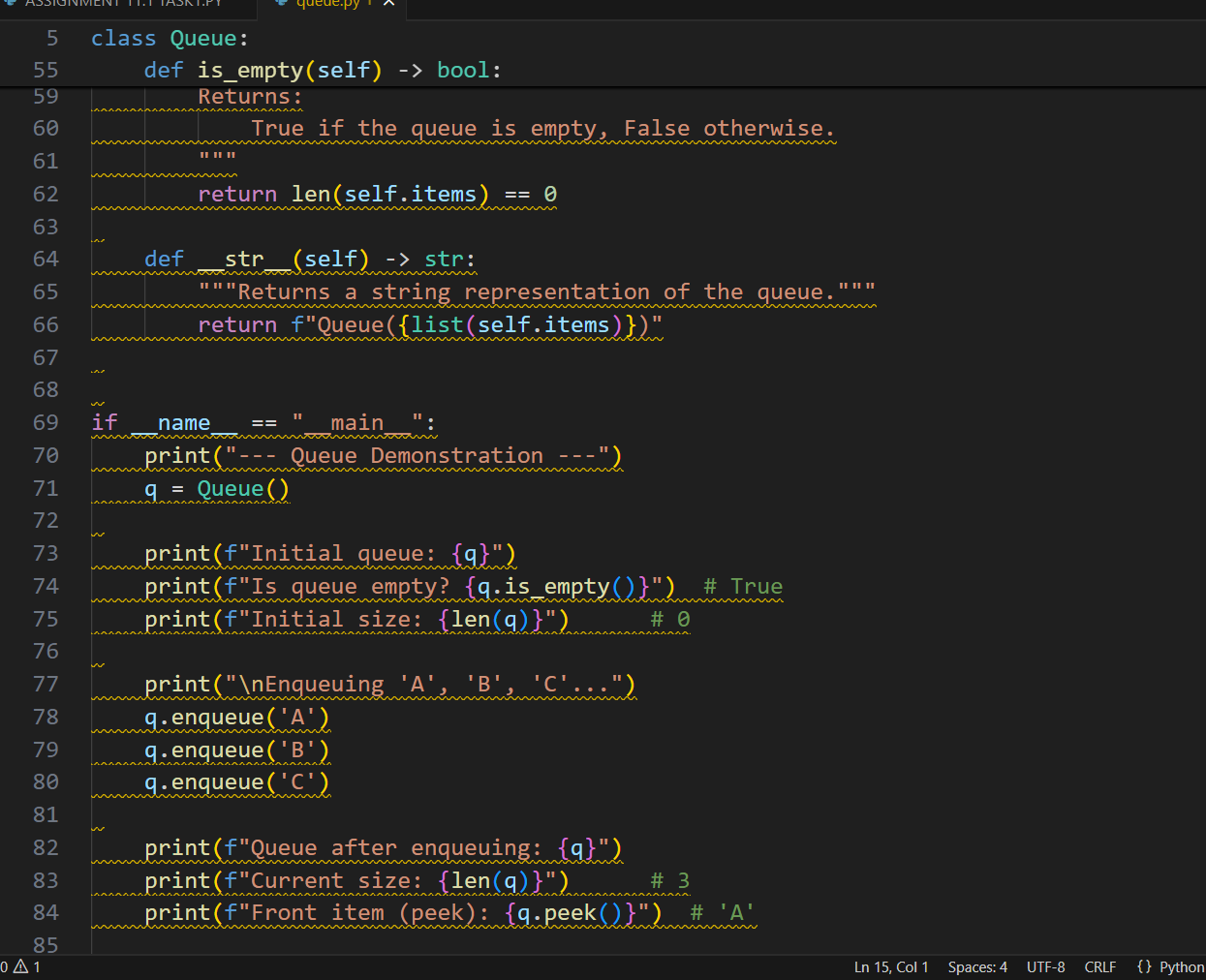
**PROMPT:**

**Generate a python code to implement a Queue using Python lists. with methods fifo-based queue class with enqueue, dequeue, peek, and size.**

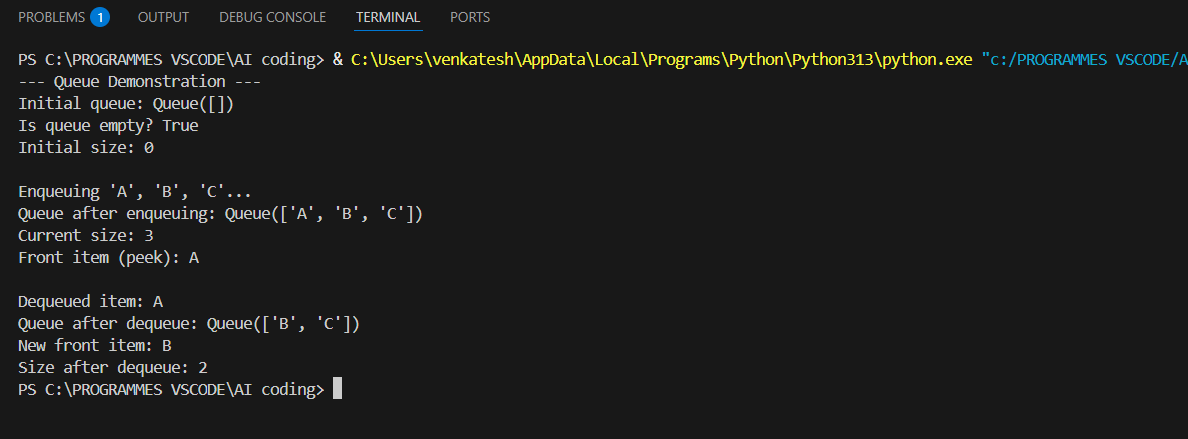
**CODE:**

****

****

****

**OUTPUT:**

****

**OBSERVATION:**

**This Python code defines a Queue class implementing a First-In, First-Out (FIFO) structure using the highly efficient collections deque for optimal performance. It provides standard methods like enqueue, dequeue, and peek for queue manipulation. The class is also made more intuitive and Pythonic by including special methods like \_len\_ and \_str\_**

## TASK-3

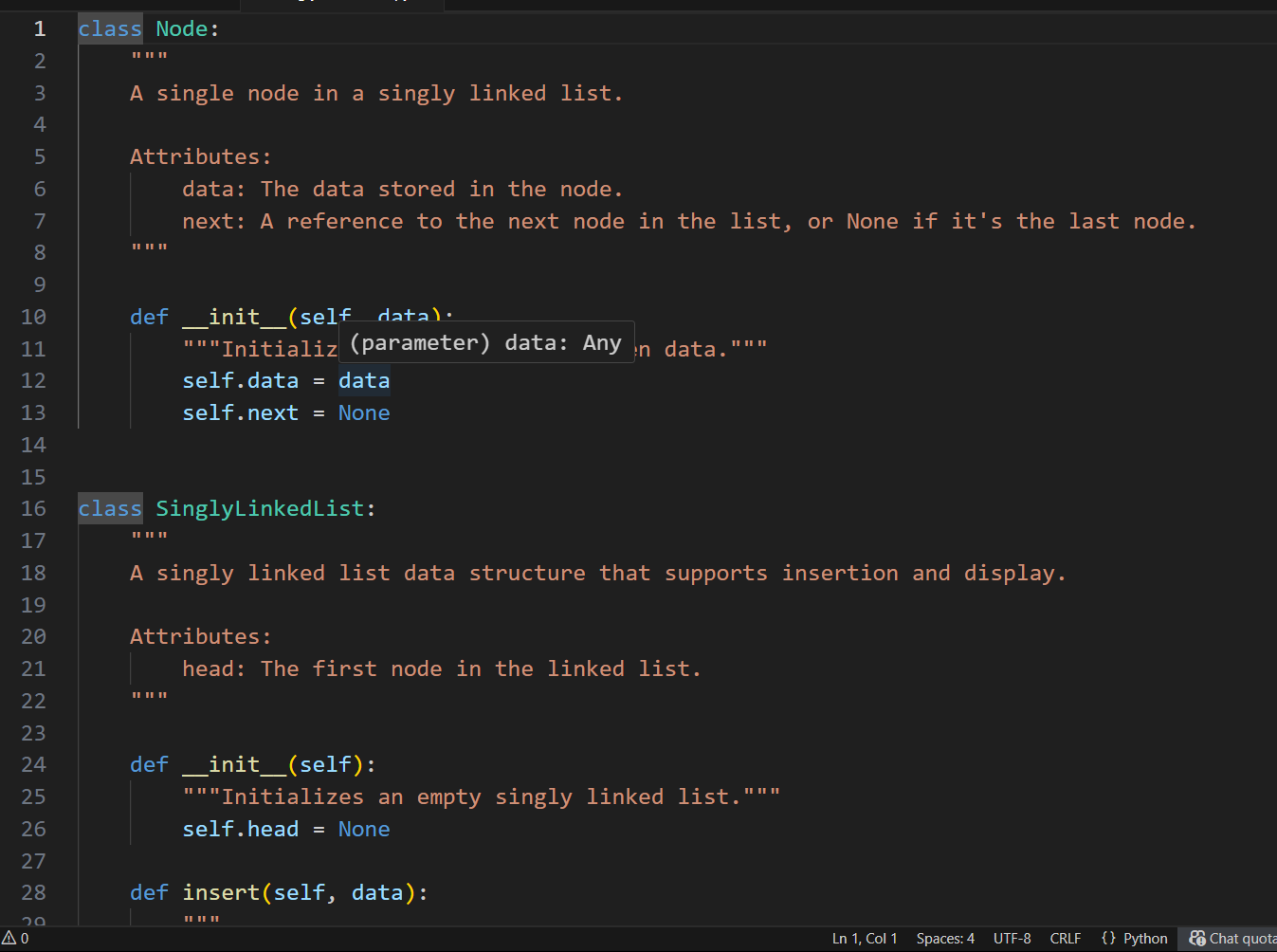
**QUESTION:**

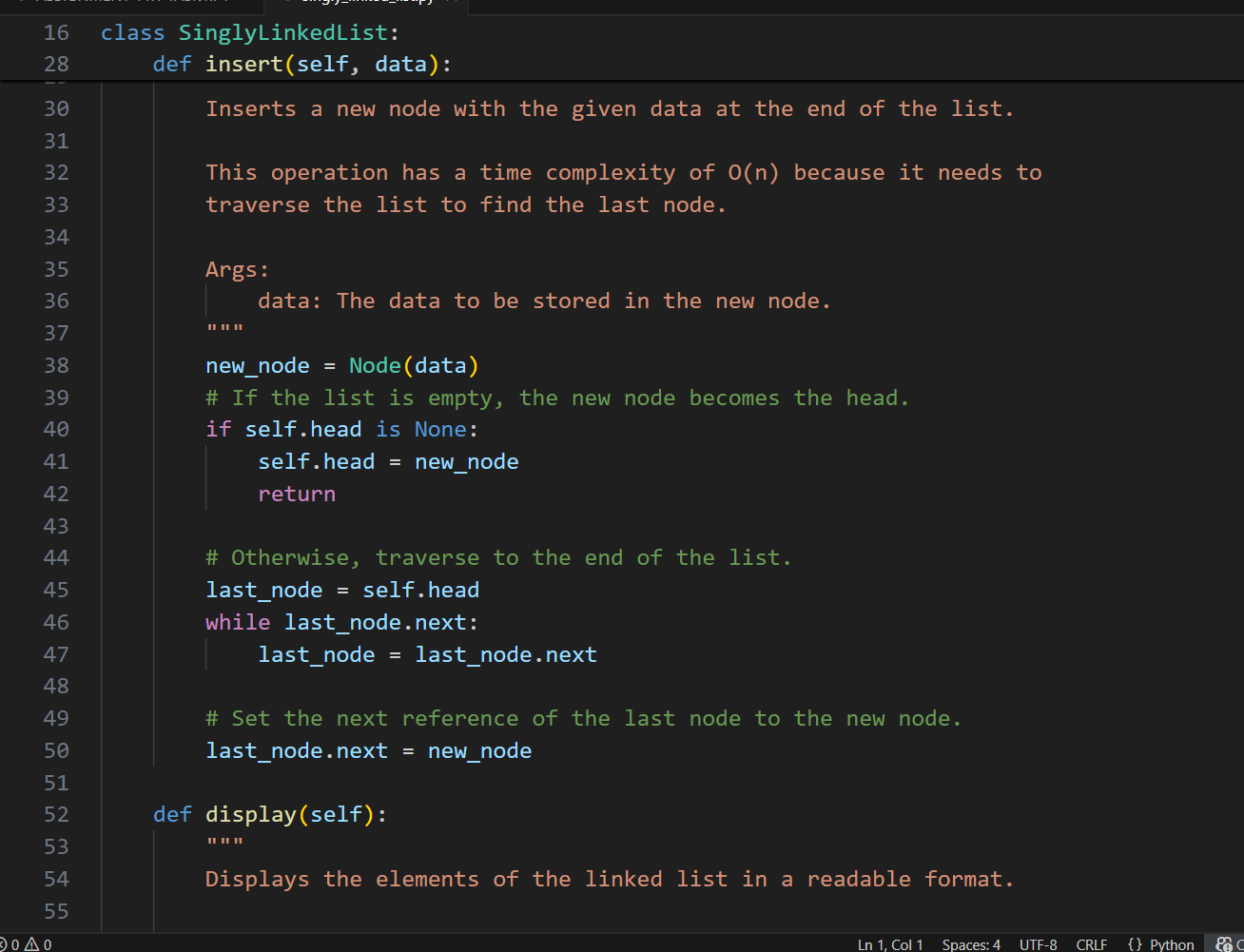
**3 – Linked List  
Task: Use AI to generate a Singly Linked List with insert and display methods.  
Sample Input Code:  
class Node:  
pass  
class LinkedList:  
pass  
Expected Output:  
• A working linked list implementation with clear method  
documentation**

**PROMPT:**

**Generate a python code Singly Linked List with insert and display methods.with clear method documentation**

**CODE:**

****

****

****

**OUTPUT:**

****

**OBSERVATION:**

**This Python code provides a clear and fundamental implementation of a Singly LinkedList with its corresponding Node class. It correctly implements an insert method that adds new nodes to the end of the list (an O(n) operation) and a display method for easy visualization. The code is well-documented with docstrings and includes a simple demonstration block, making it an excellent example for learning this data structure.**

## TASK-4

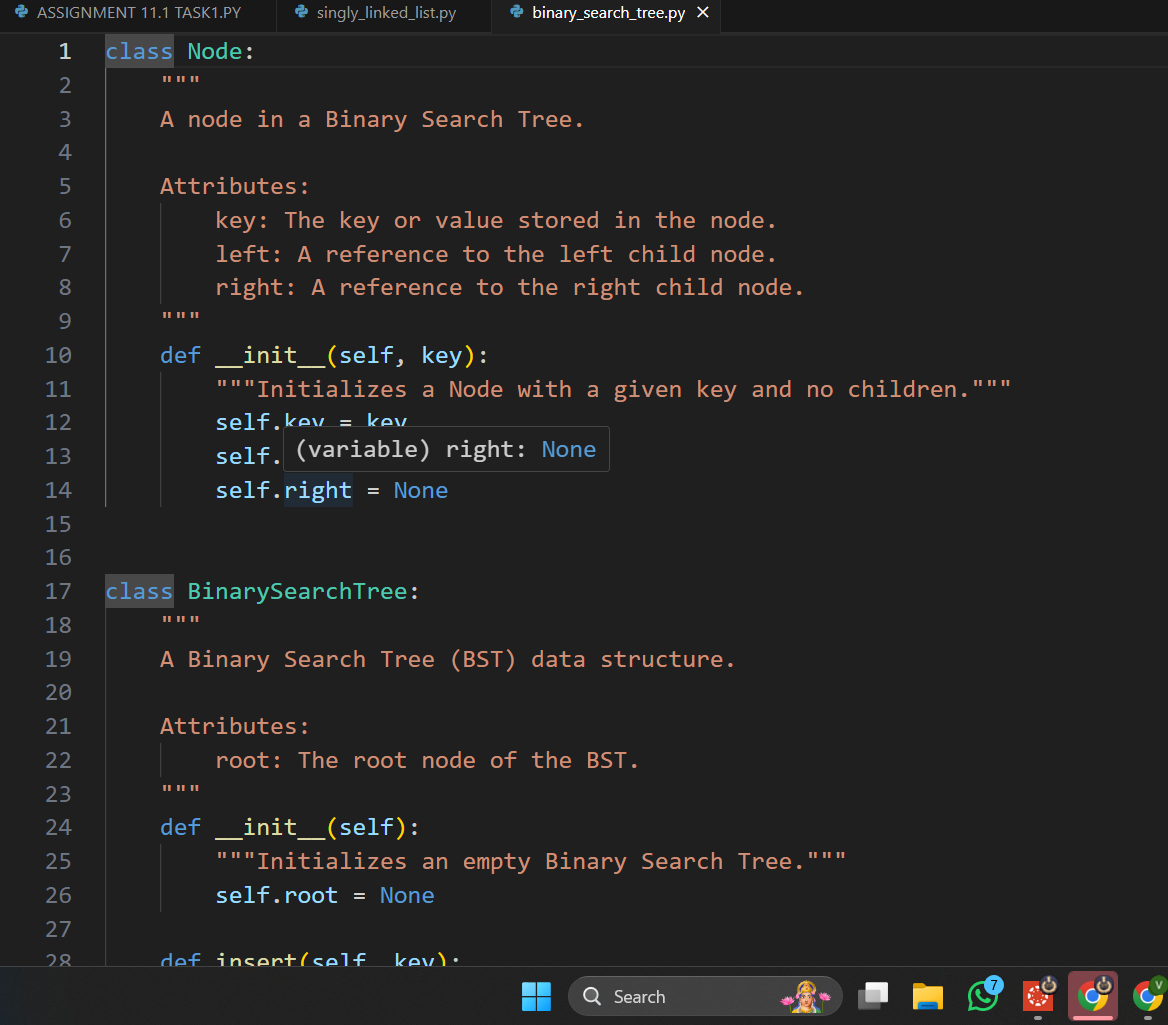
**QUESTION:**

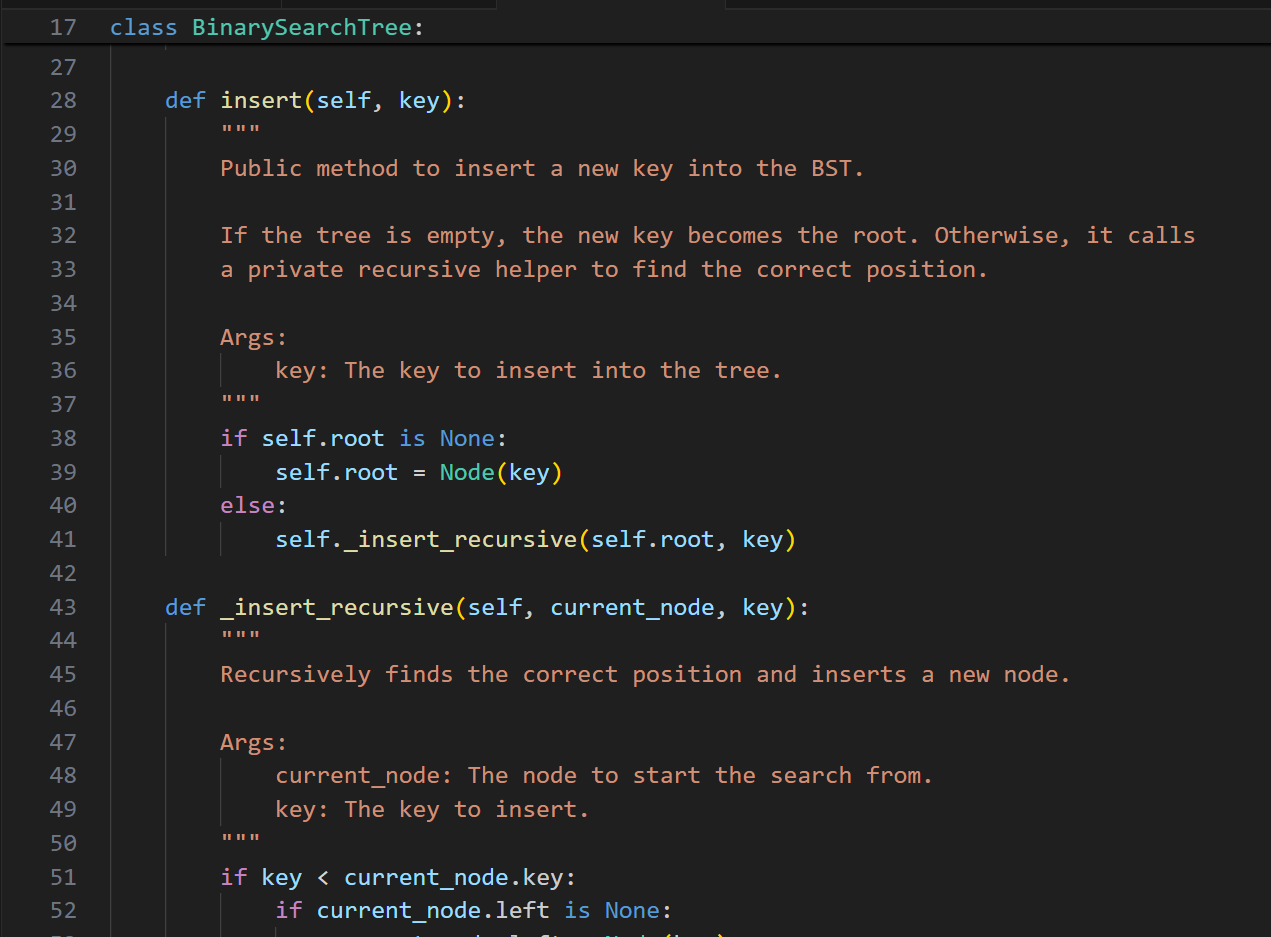
**4 – Binary Search Tree (BST)  
Task: Use AI to create a BST with insert and in-order traversal methods.  
Sample Input Code:  
class BST:  
pass  
Expected Output:  
• BST implementation with recursive insert and traversal methods**

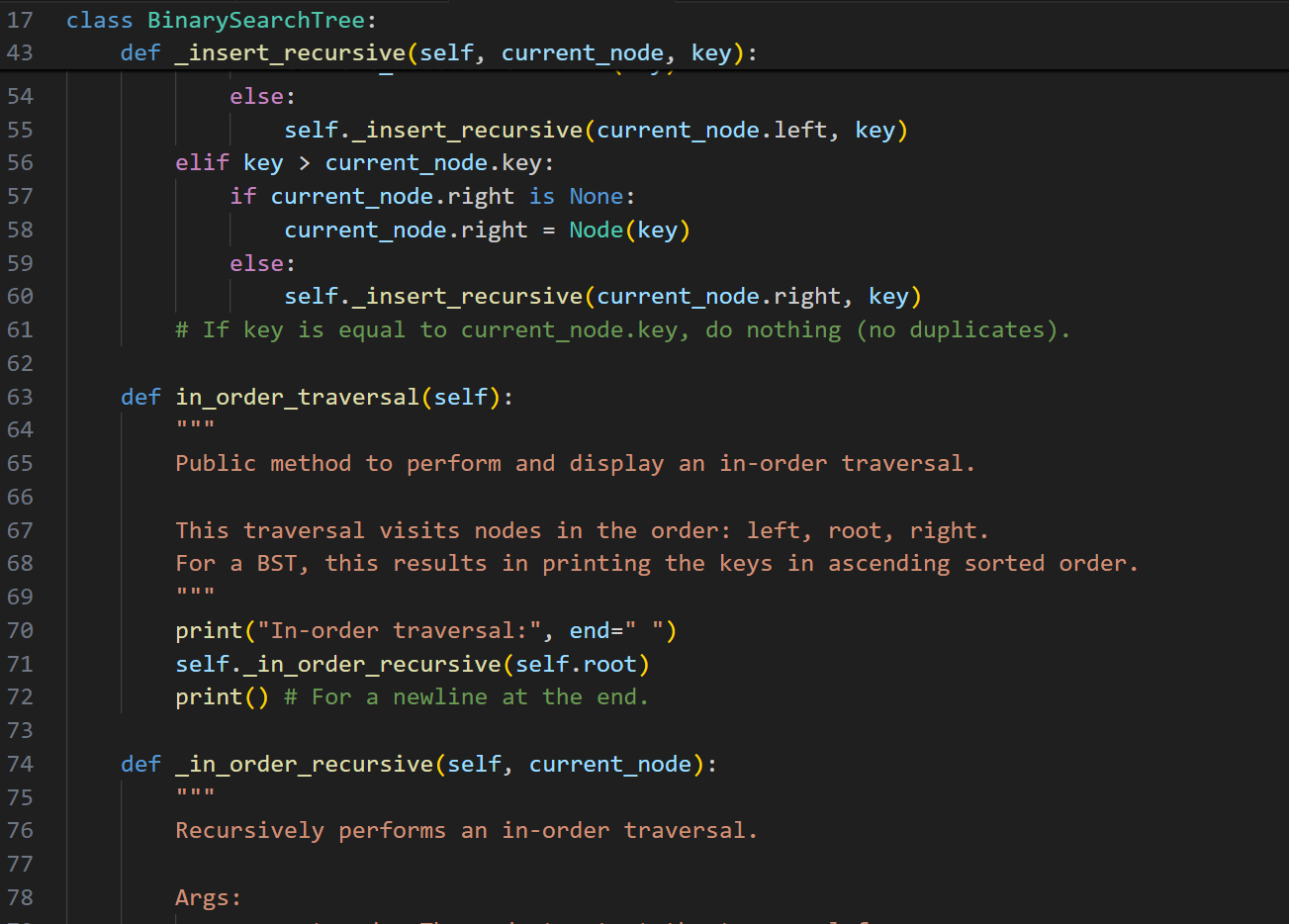
**PROMPT:**

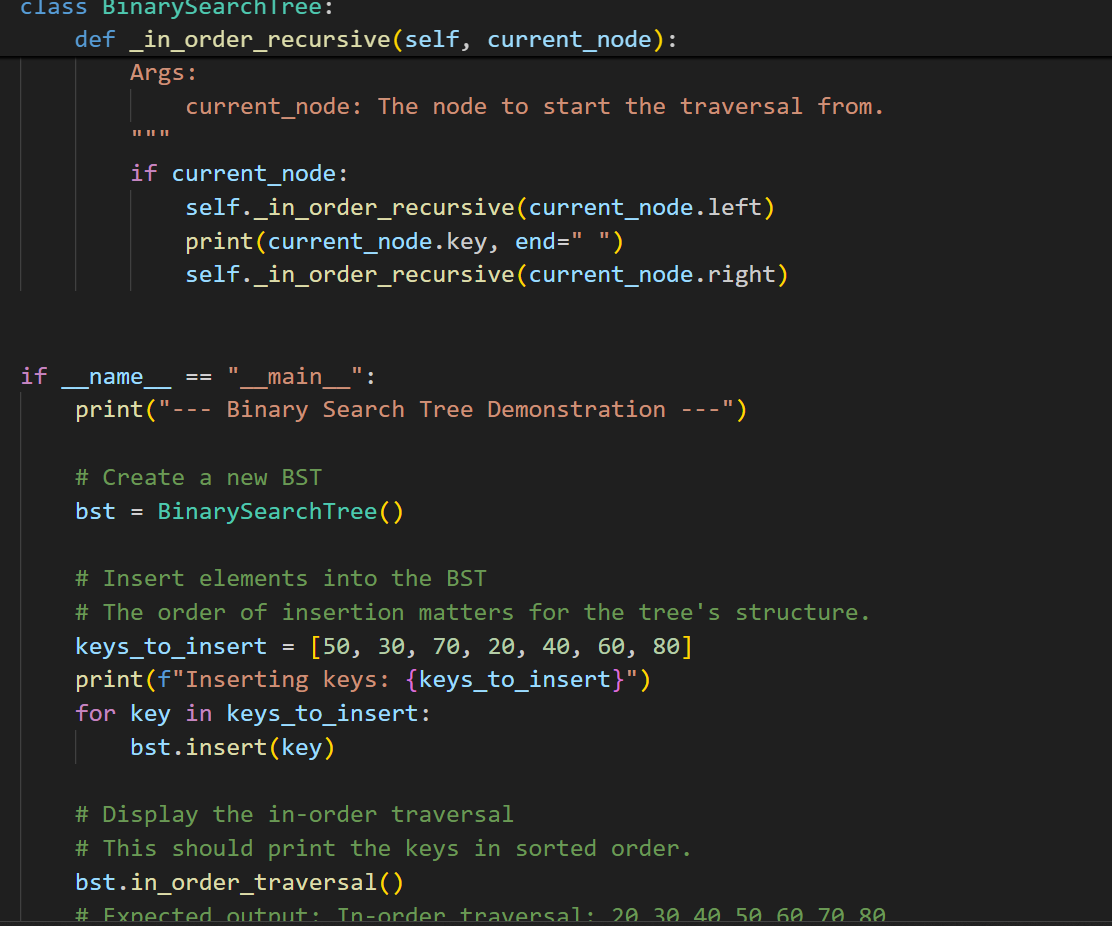
**create a python code BST with insert and in-order traversal methods**

**CODE:**

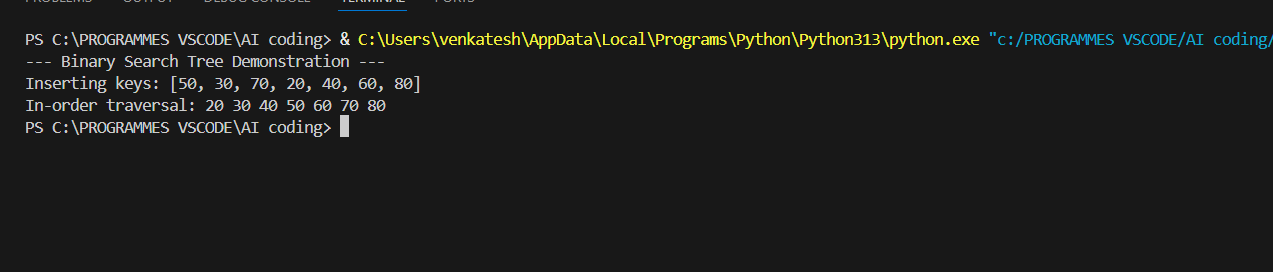
****

****

****

****

**OUTPUT:**

****

**OBSESRVATION:**

**This code provides a clean, recursive implementation of a Binary Search Tree with insert and in order traversal methods. The structure is well-defined with separate Node and Binary Search Tree classes, and the traversal correctly yields the elements in sorted order.**

## TASK-5

**QUESTION:**

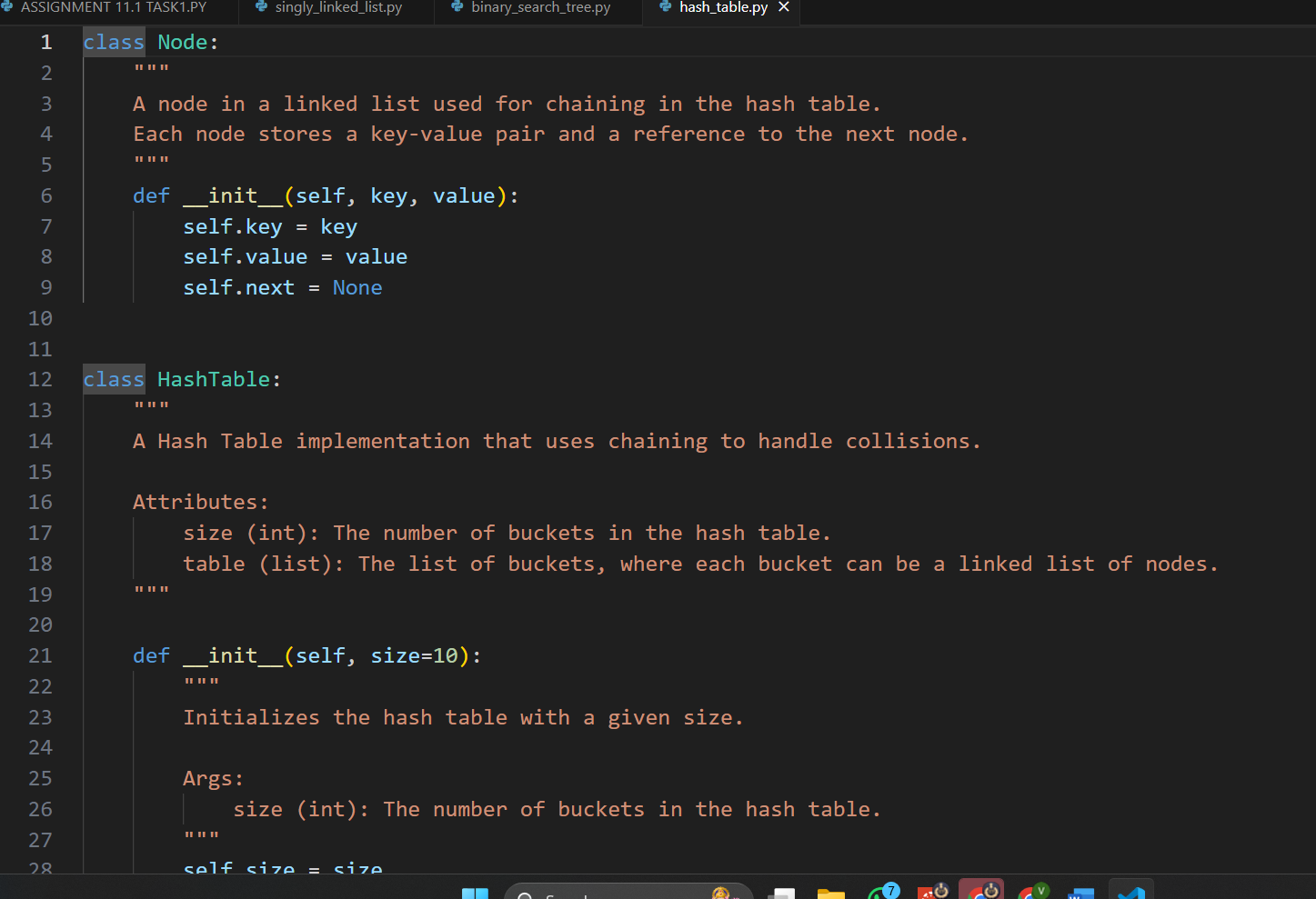
**5– Hash Table  
Task: Use AI to implement a hash table with basic insert, search, and delete**

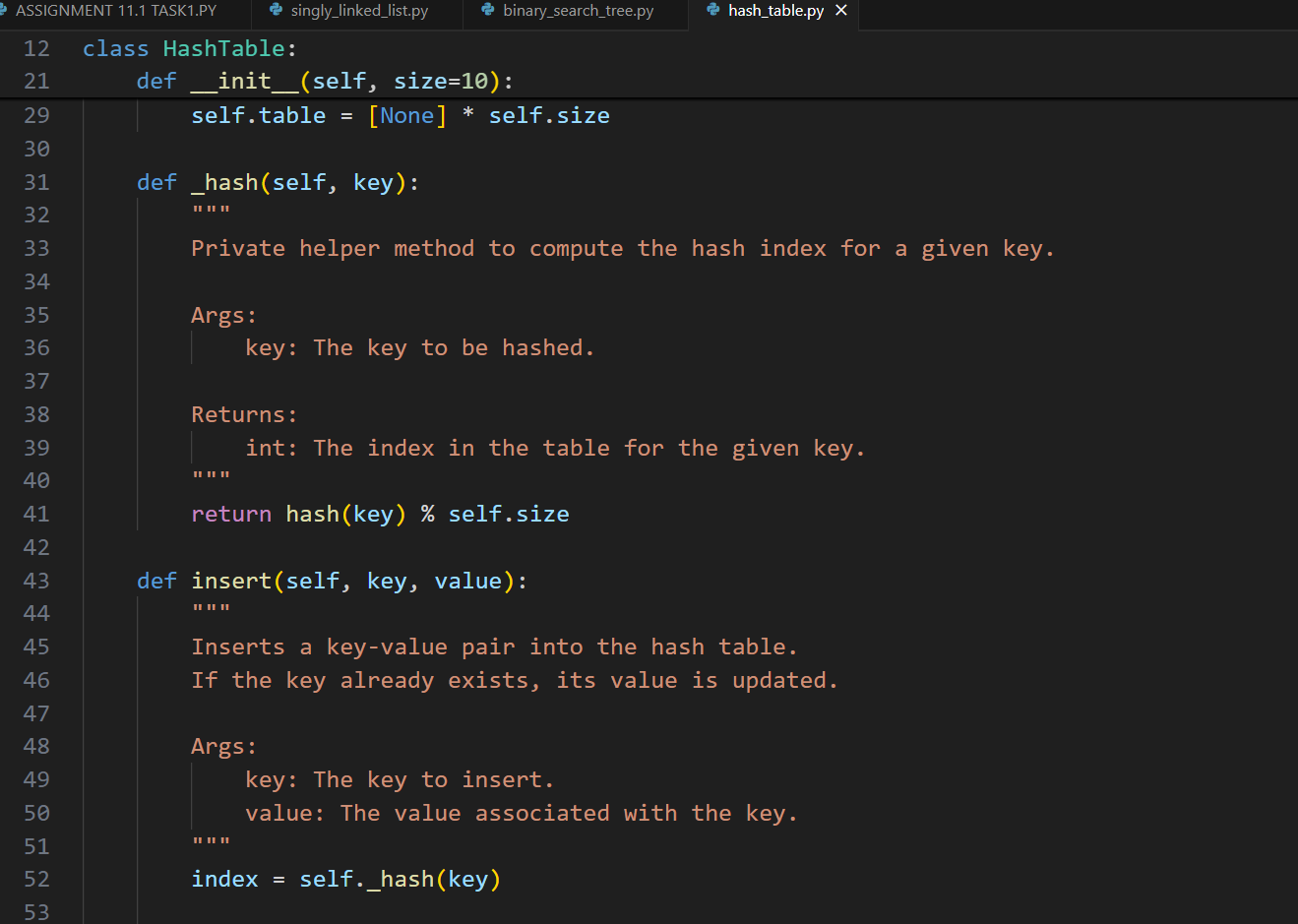
**methods.  
Sample Input Code:  
class HashTable:  
pass  
Expected Output:  
• Collision handling using chaining, with well-commented methods**

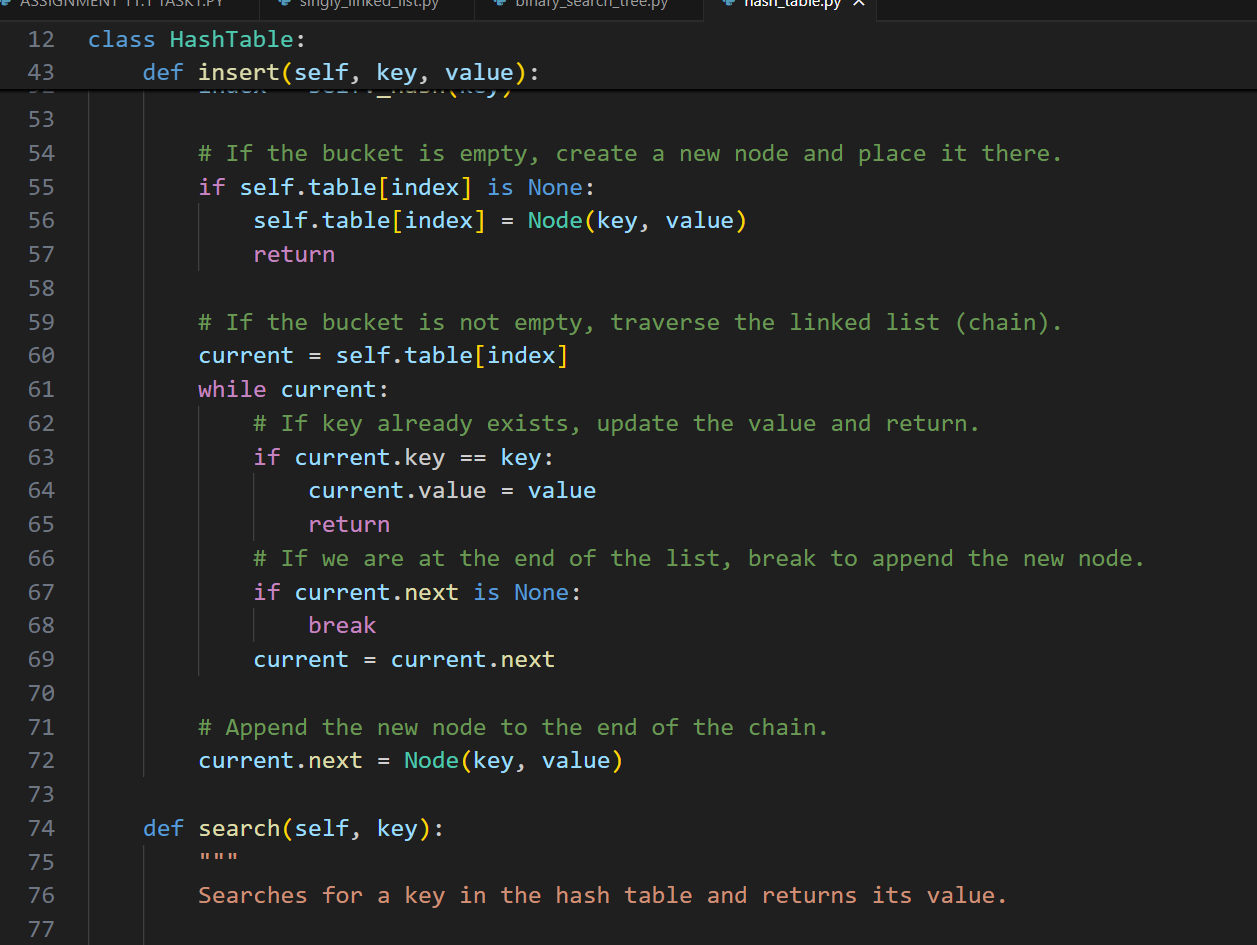
**PROMPT:**

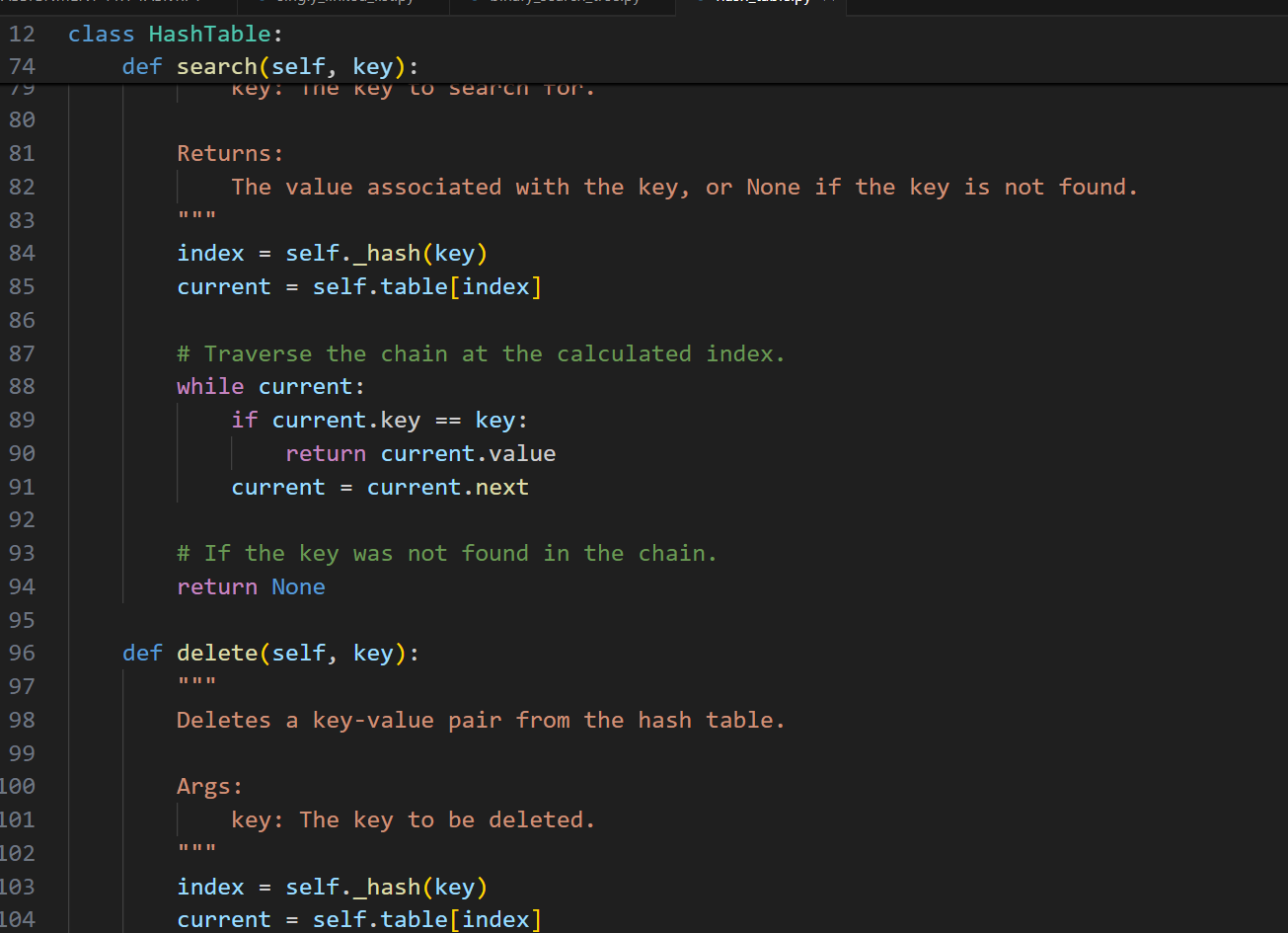
**Impliment a python code ash table with basic insert, search, and delete methods and Collision handling using chaining, with well-commented methods**

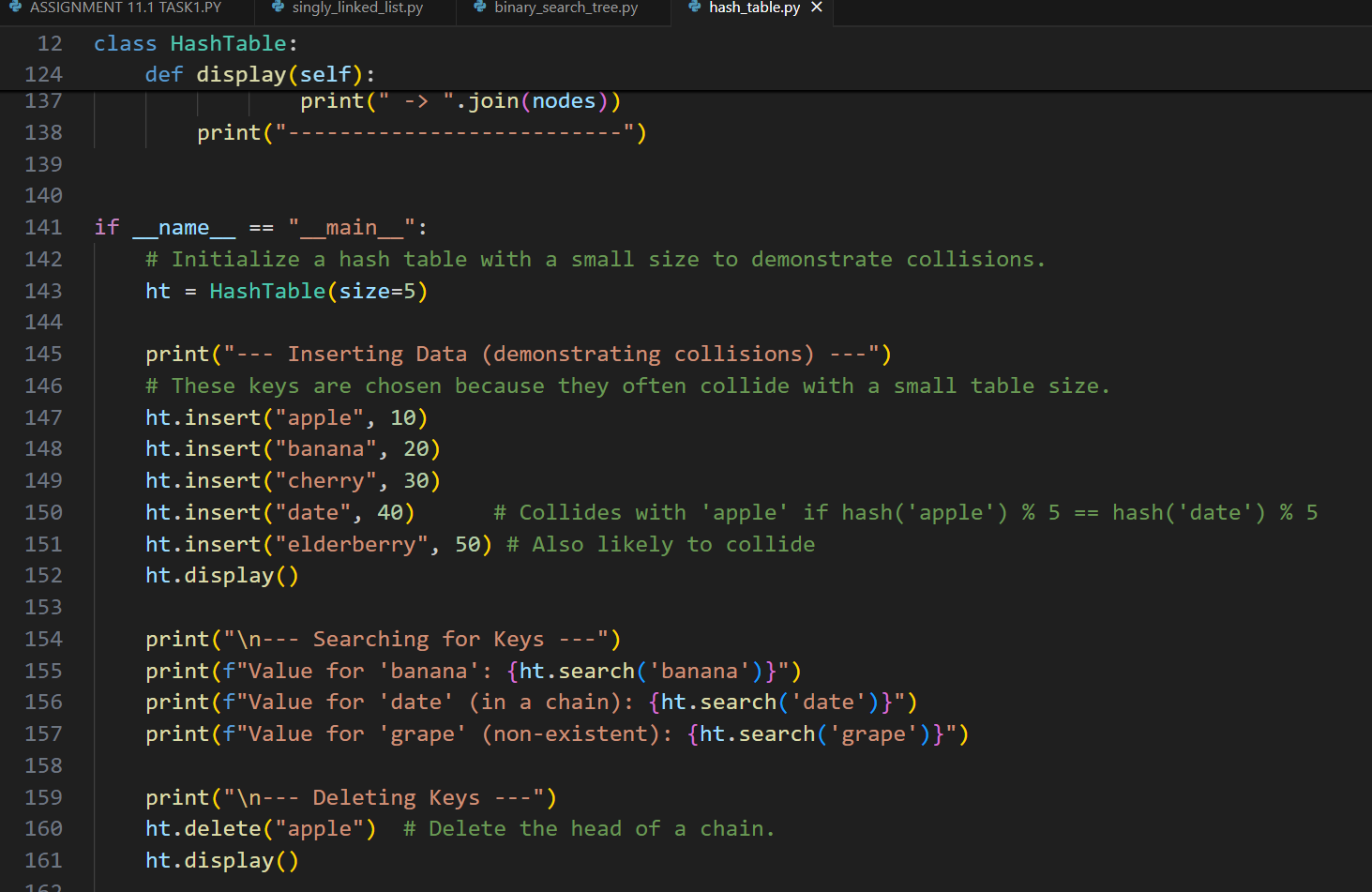
**CODE:**

****

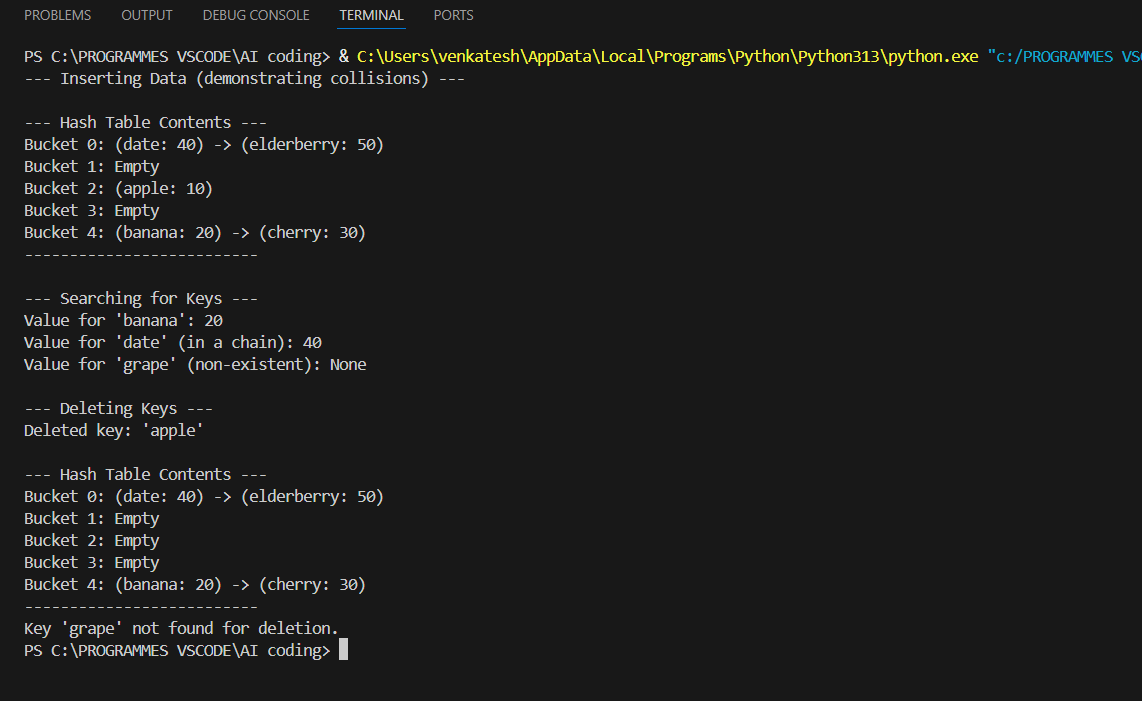
****

****

****

****

**OUTPUT:**

****

**OBSERVATION:**

**This code provides a robust implementation of a Hash Table using the chaining method with a linked list to handle collisions. It correctly implements the essential insert, search, and delete operations, including logic for updating existing keys and handling deletions within the chains. The code is well-commented and includes a display method and a demonstration block, making it a clear and effective educational example.**

## TASK-6

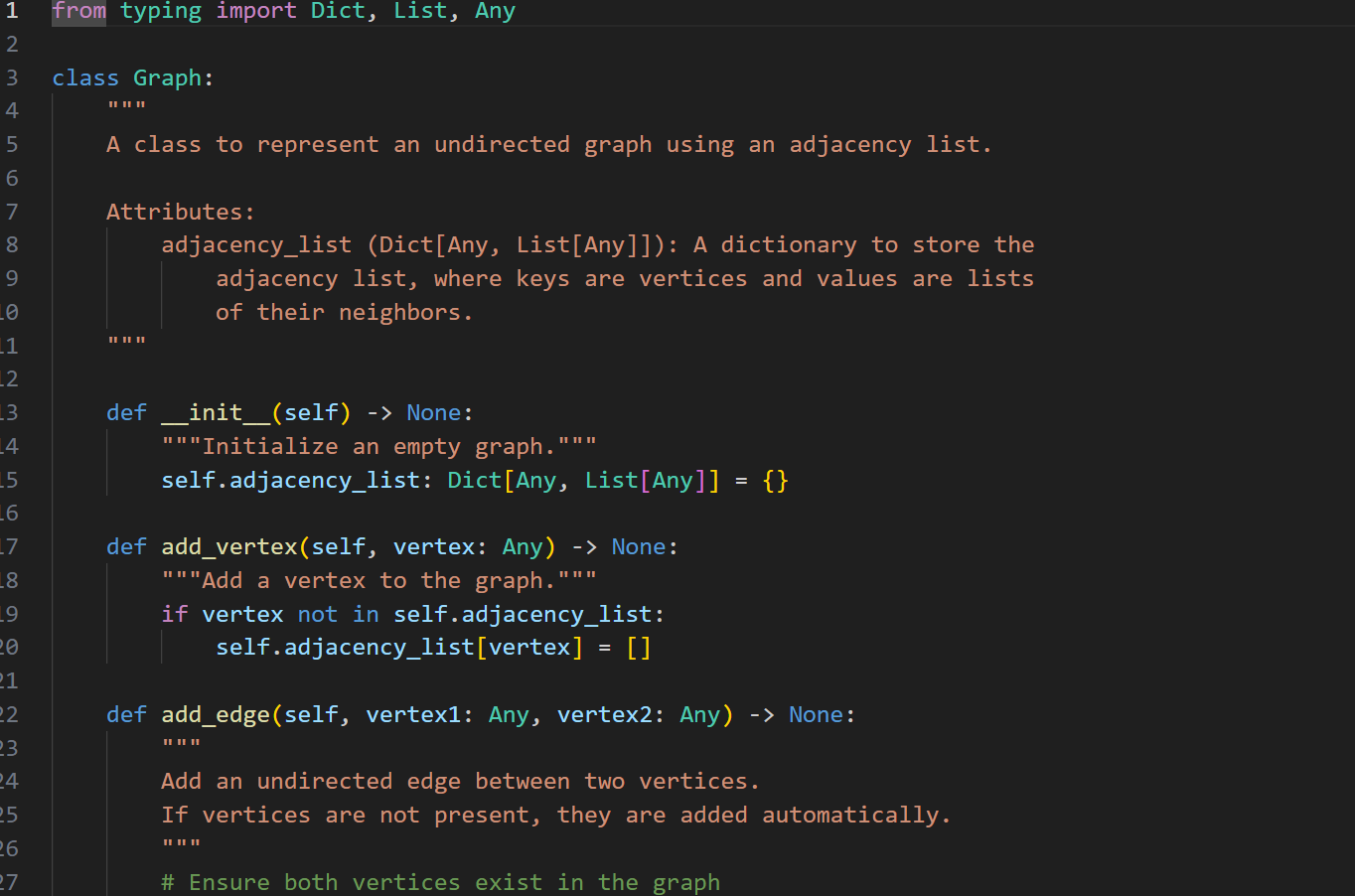
**QUESTION:**

**6 – Graph Representation  
Task: Use AI to implement a graph using an adjacency list.  
Sample Input Code:  
class Graph:  
pass  
Expected Output:  
• Graph with methods to add vertices, add edges, and display  
connections.**

**PROMPT:**

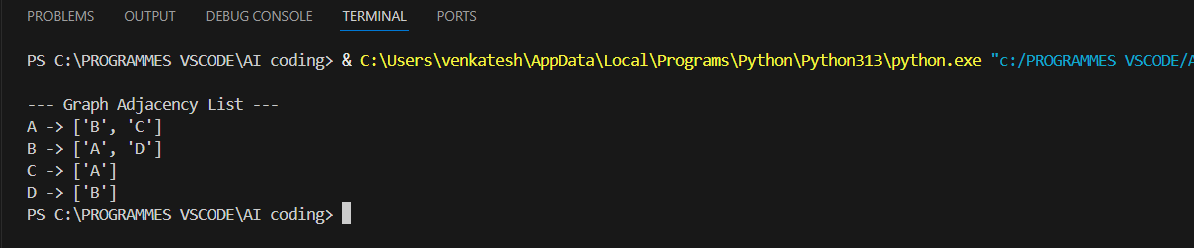
**Write a Python class Graph using an adjacency list. It should have methods to add a vertex, add an edge (undirected), and display the graph. Use a dictionary to store the adjacency list and add a small example.**

**CODE:**

****

****

**OUTPUT:**

****

**OBSERVATION:**

**This Python code provides a clear and effective implementation of an undirected graph using a dictionary-based adjacency list, which is a standard representation**

## TASK-7

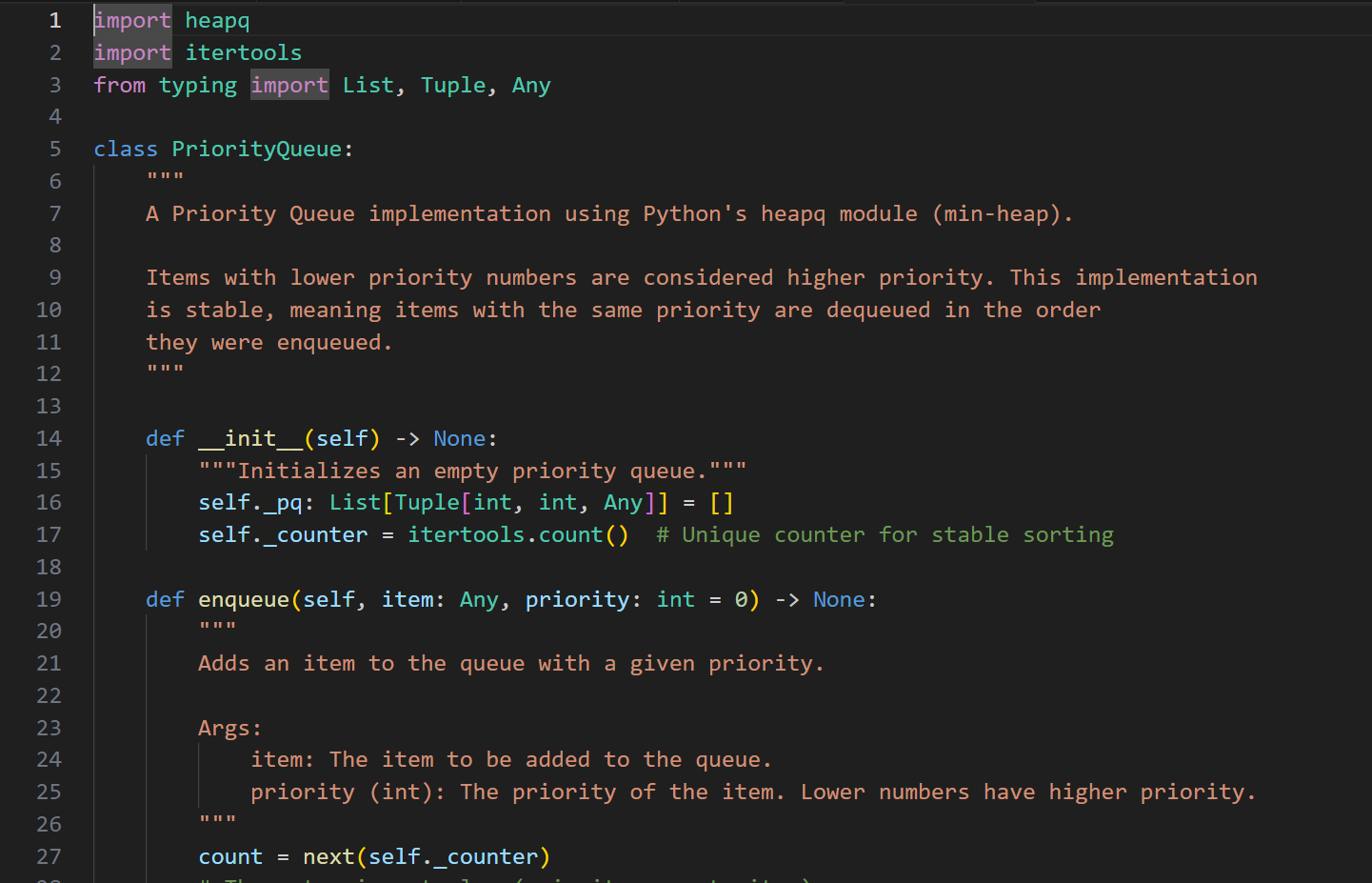
**QUESTION:**

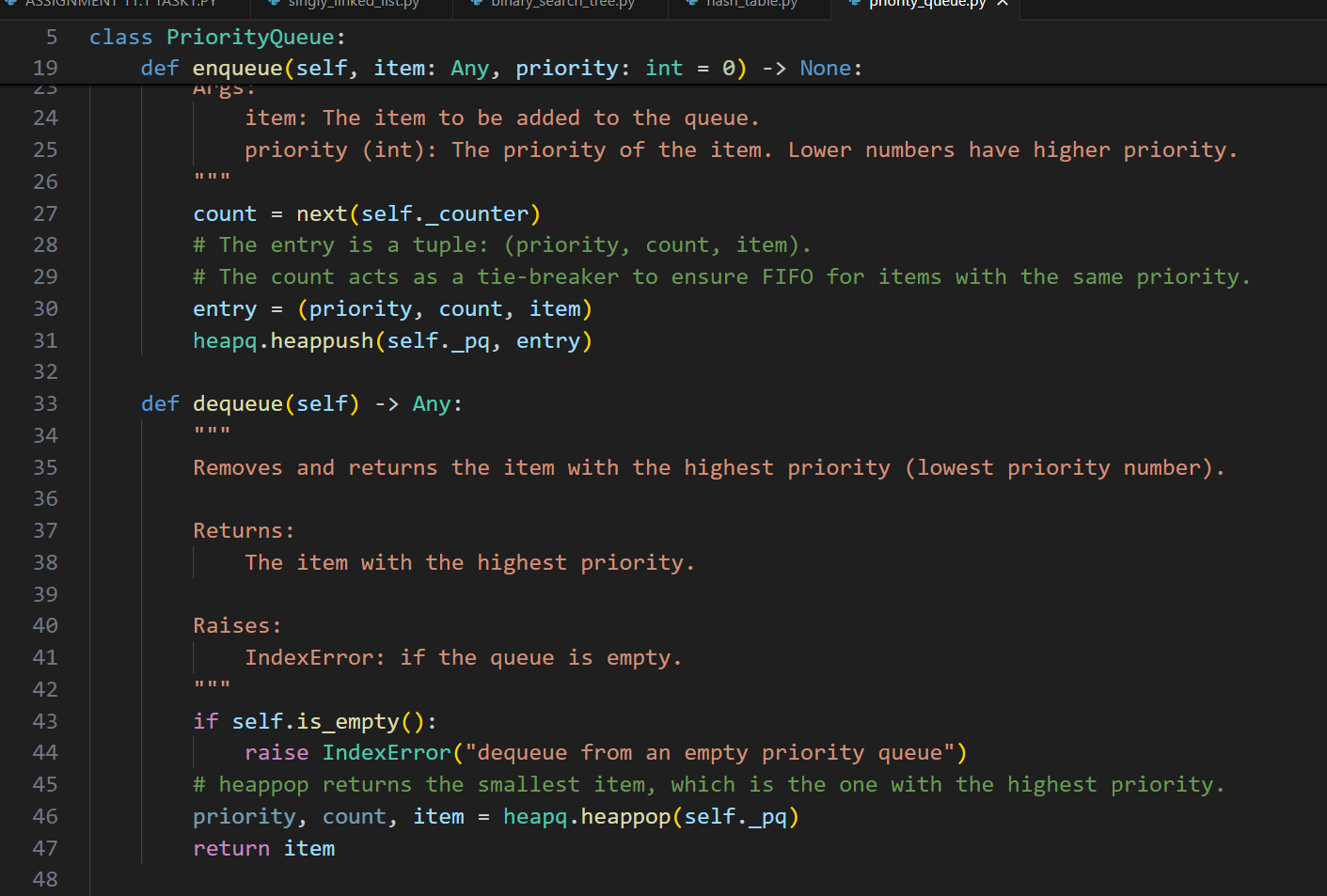
**7 – Priority Queue  
Task: Use AI to implement a priority queue using Python’s heapq module.  
Sample Input Code:  
class PriorityQueue:  
pass  
Expected Output:  
• Implementation with enqueue (priority), dequeue (highest priority), and  
display methods**

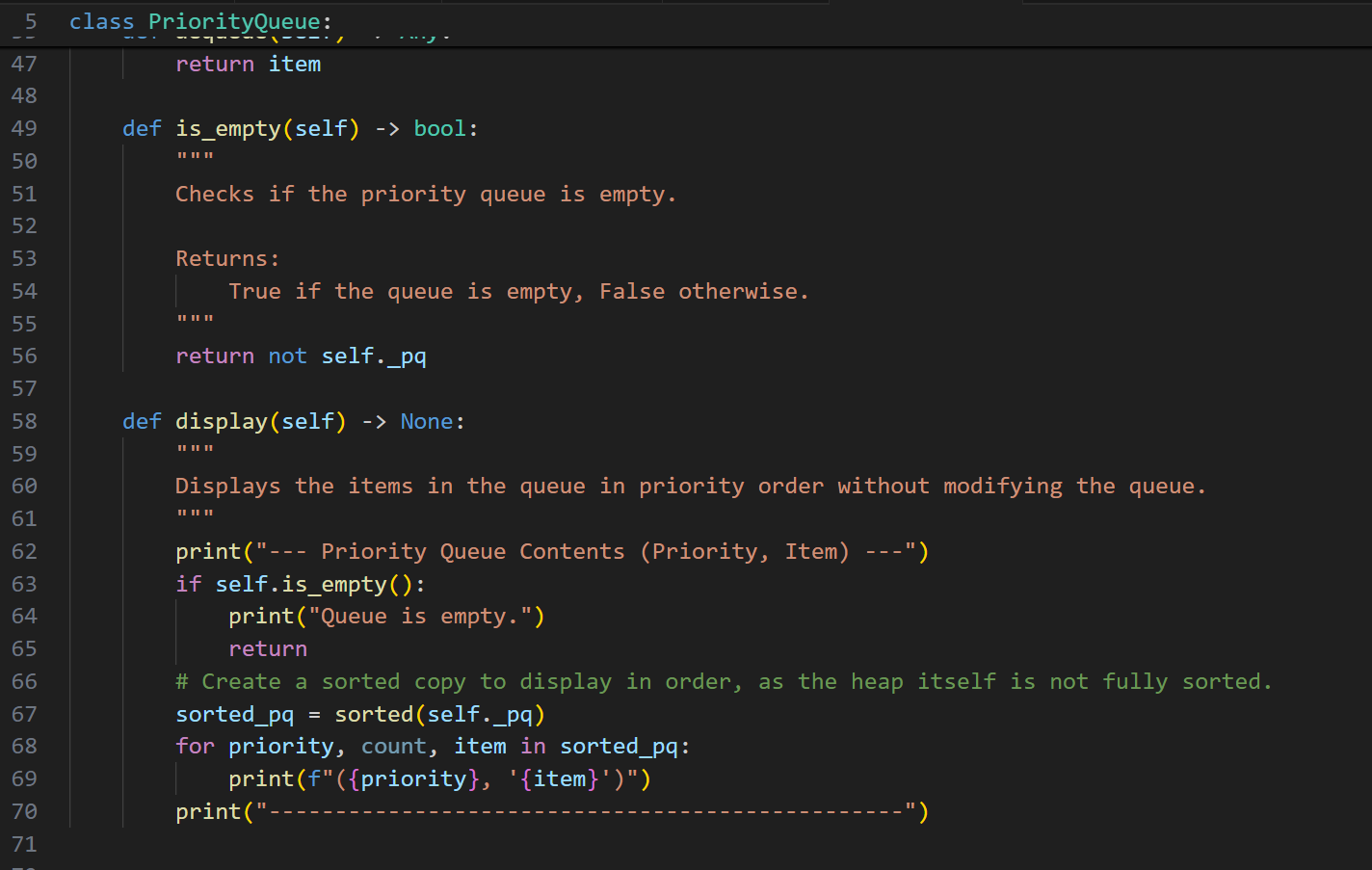
**PROMPT:**

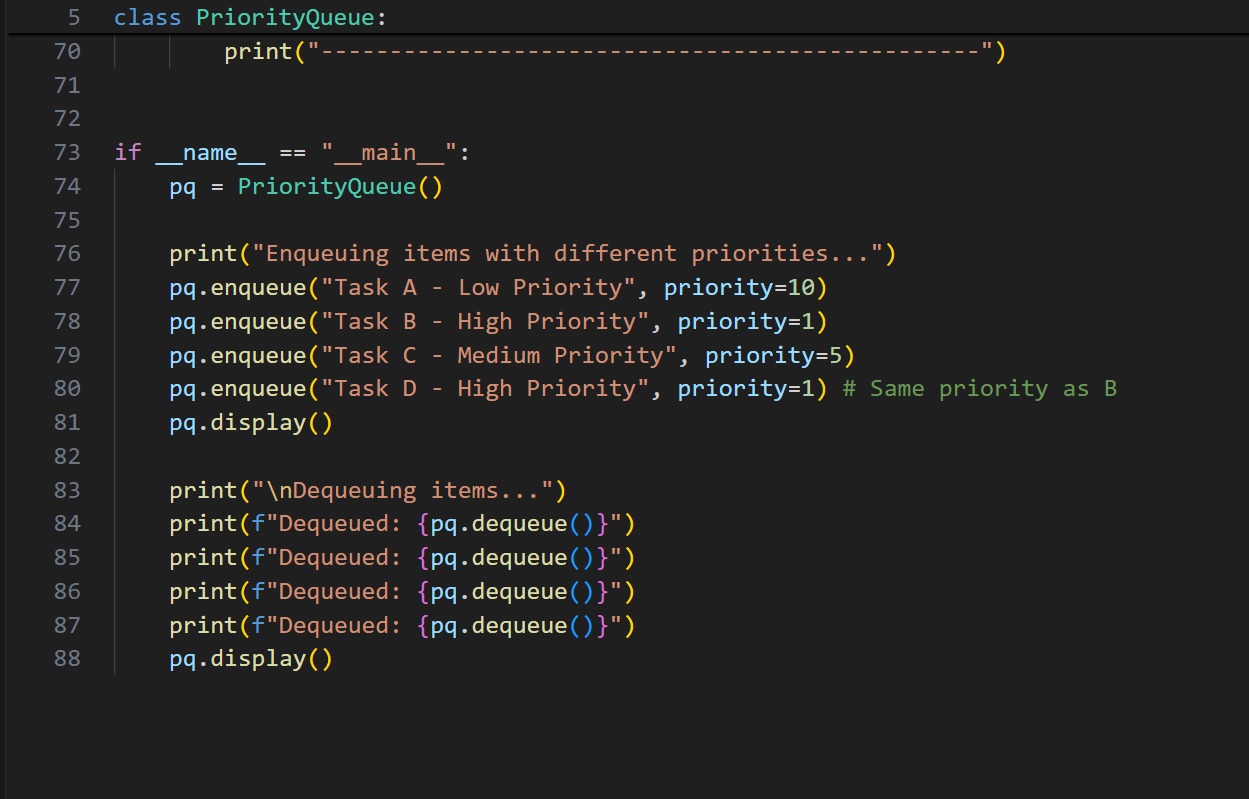
**create a python code by implement a priority queue using Python’s heapq module.with nqueue (priority), dequeue (highest priority), and display methods**

**CODE:**

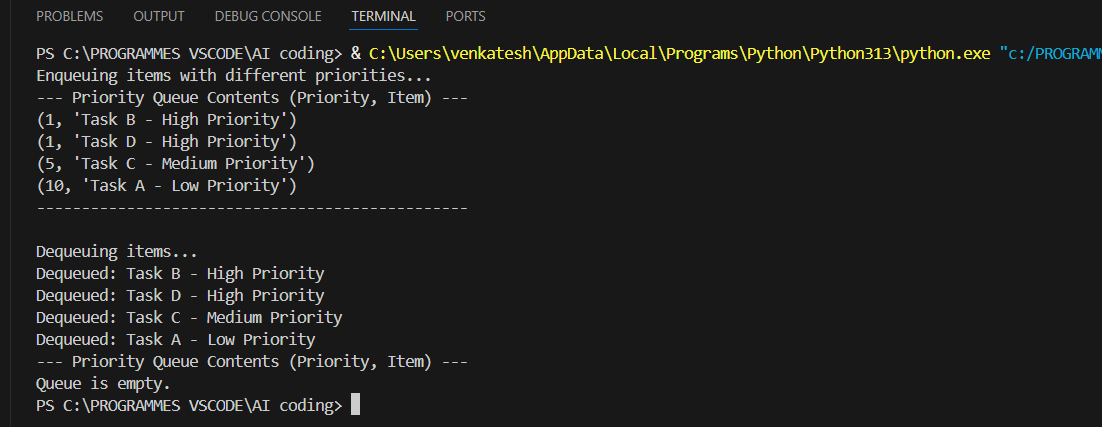
****

****

****

****

**OUTPUT:**

****

**OBSERVATION:**

**This is a high-quality Priority Queue implementation that correctly uses Python's heapq module for efficient O(log n) operations. It cleverly employs itertools. count() as a tie-breaker to ensure stable, FIFO behavior for items sharing the same priority level. The code is clean, well-documented, and includes a clear demonstration of its functionality.**

## TASK-8

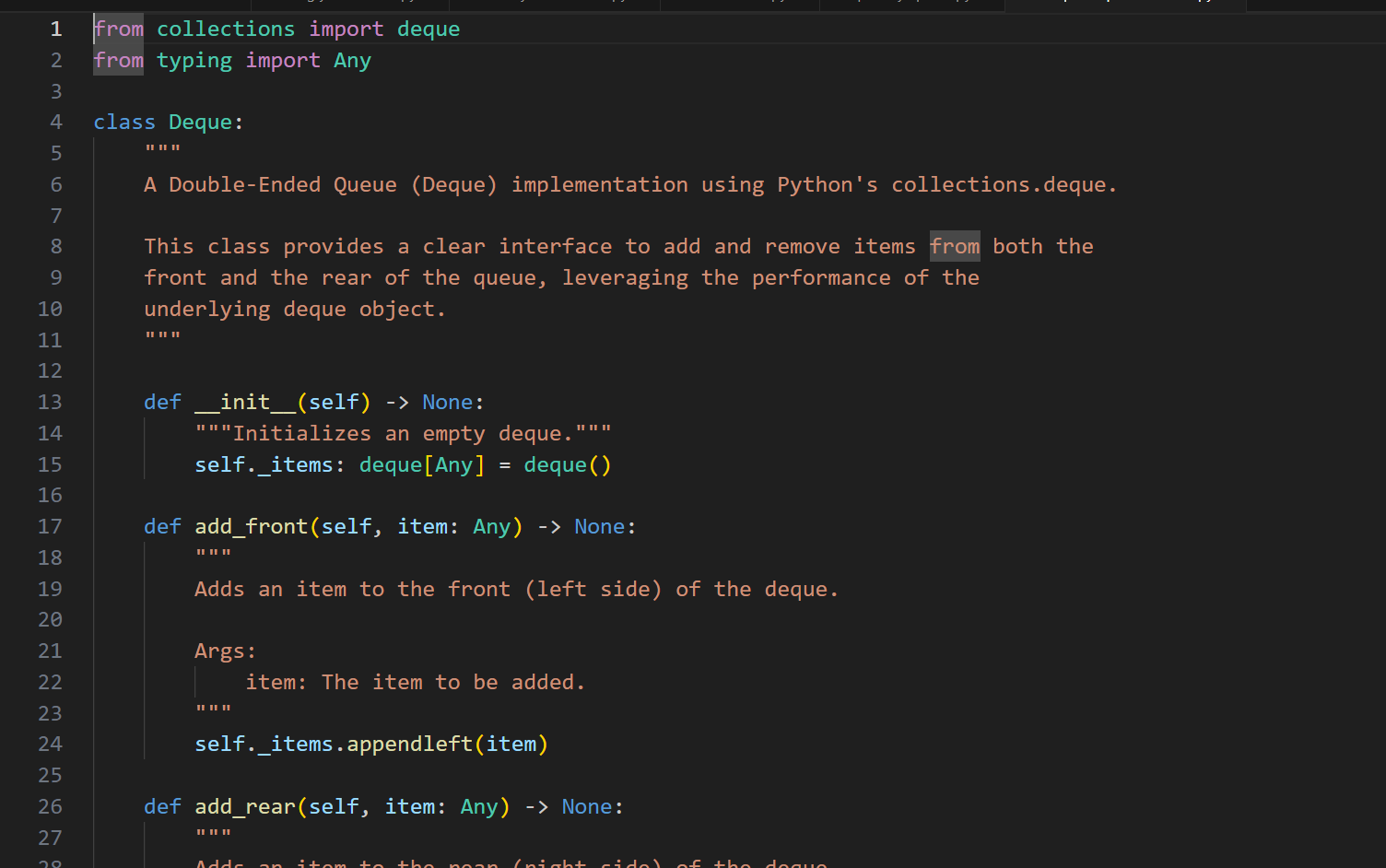
**QUESTION:**

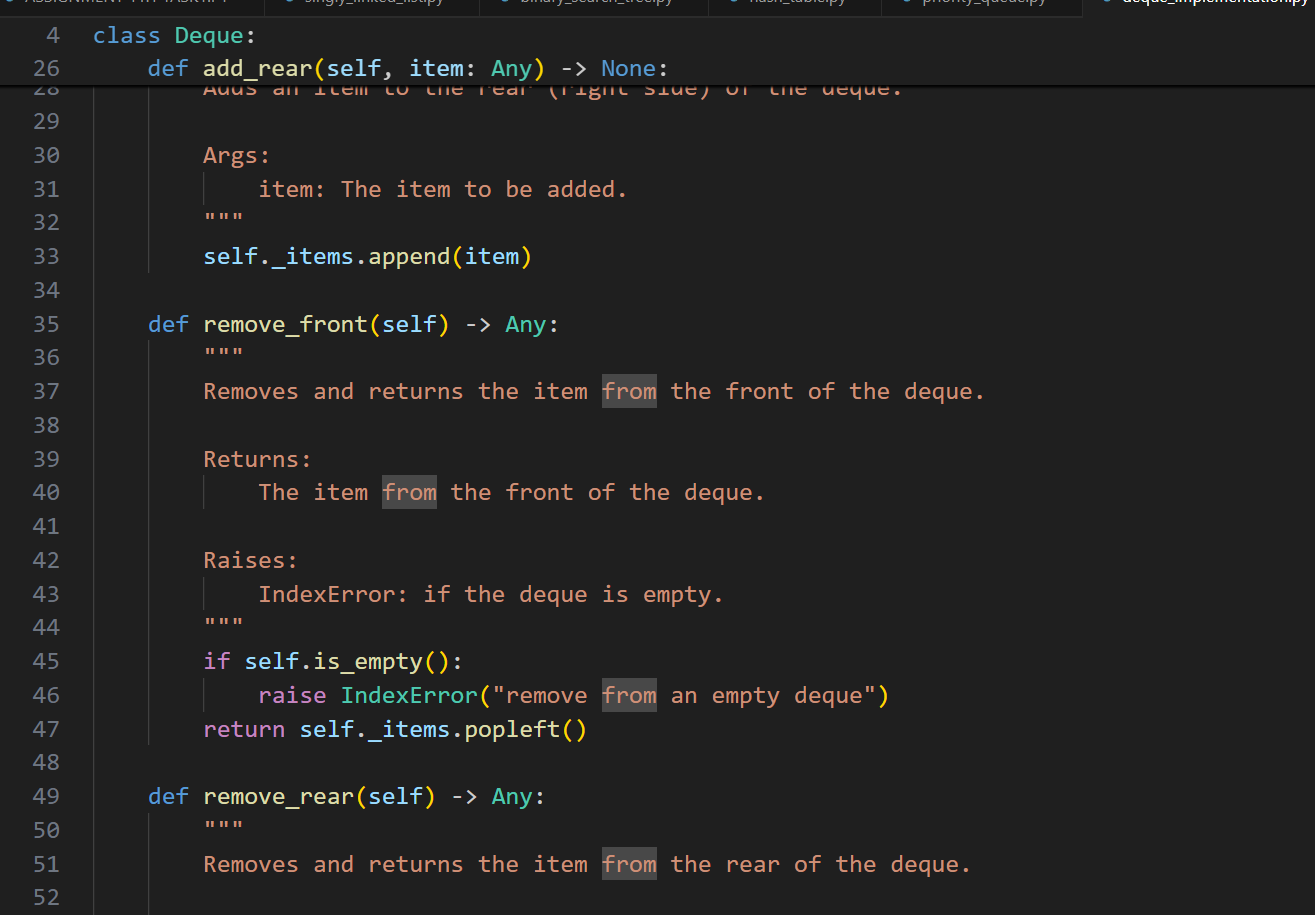
**8 – Deque  
Task: Use AI to implement a double-ended queue using collections.deque.  
Sample Input Code:  
class DequeDS:  
pass  
Expected Output:  
• Insert and remove from both ends with docstrings**

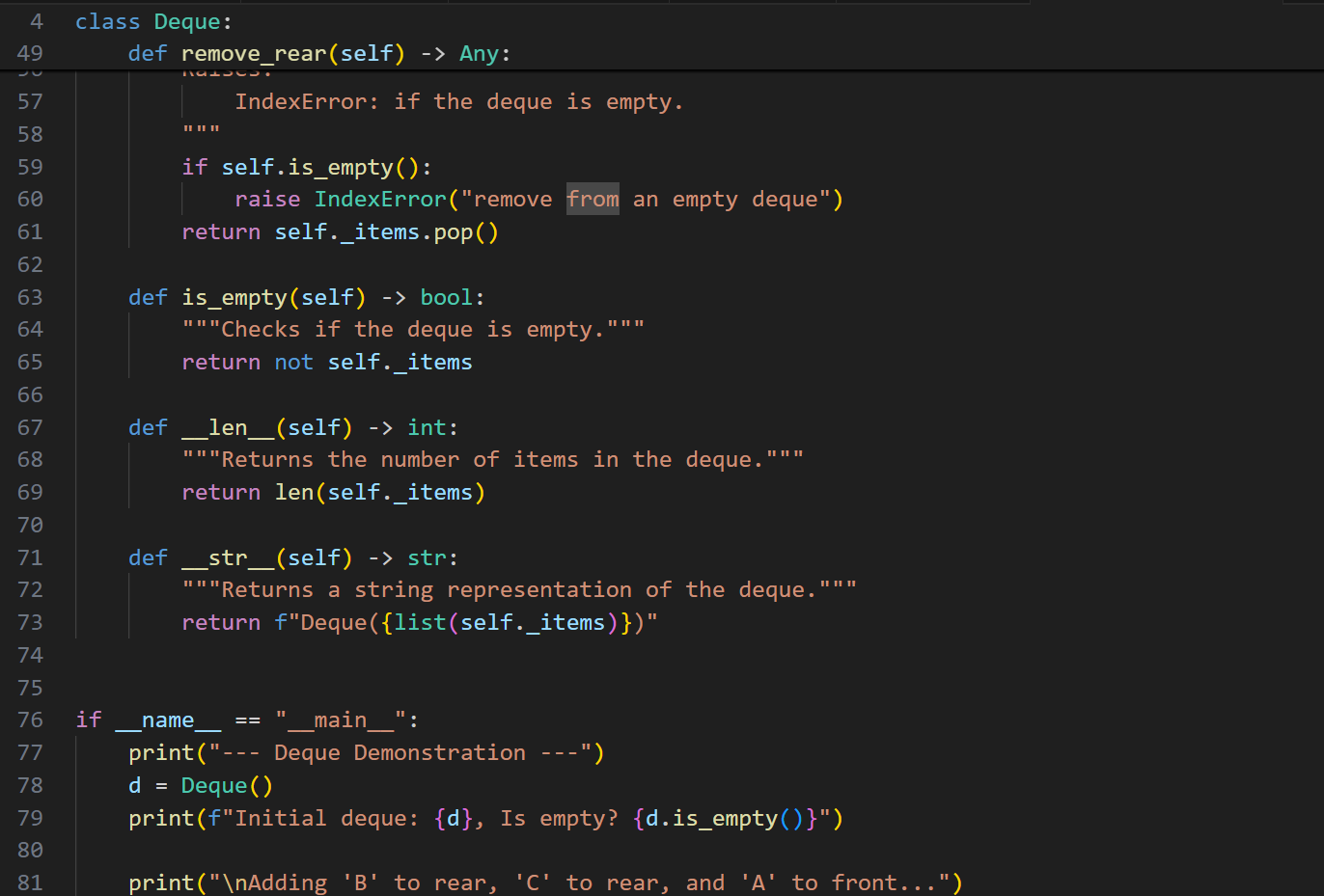
**PROMPT:**

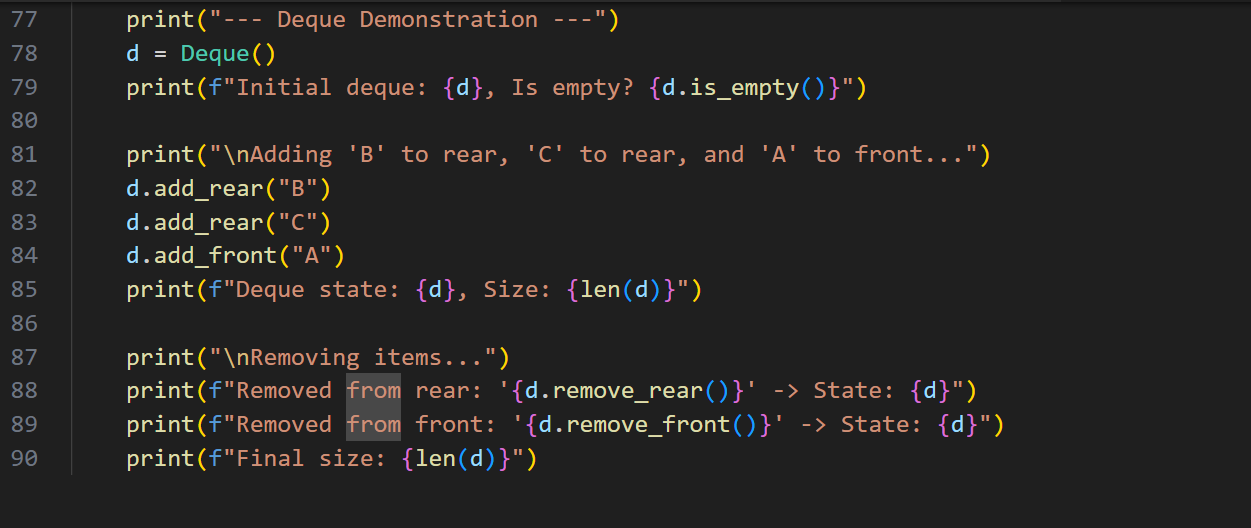
**generate a python code by mplement a double-ended queue using collections.deque with Insert and remove from both ends with docstrings.**

**CODE:**

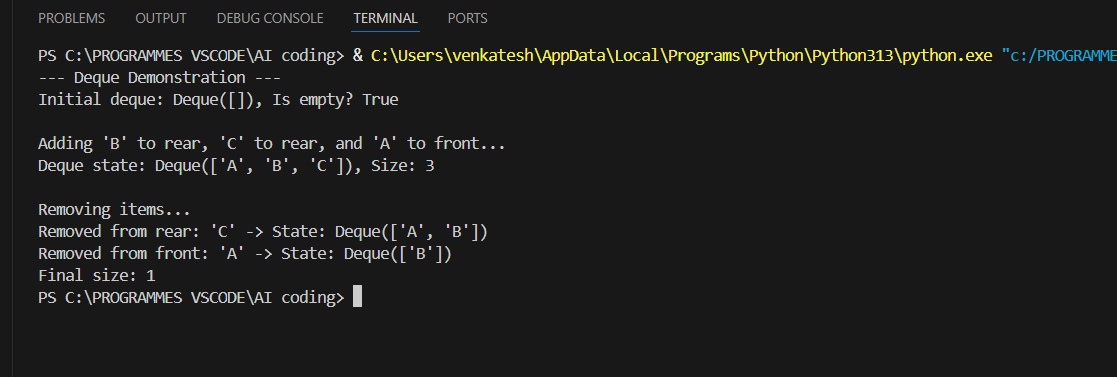
****

****

****

****

**OUTPUT:**

****

**OBSERVATION:**

**This code provides an excellent wrapper around Python's collections.deque, creating a clear and explicit Deque class with a well-defined interface. It correctly leverages the underlying deque for highly efficient O(1) appends and pops from both ends.**

## TASK-9

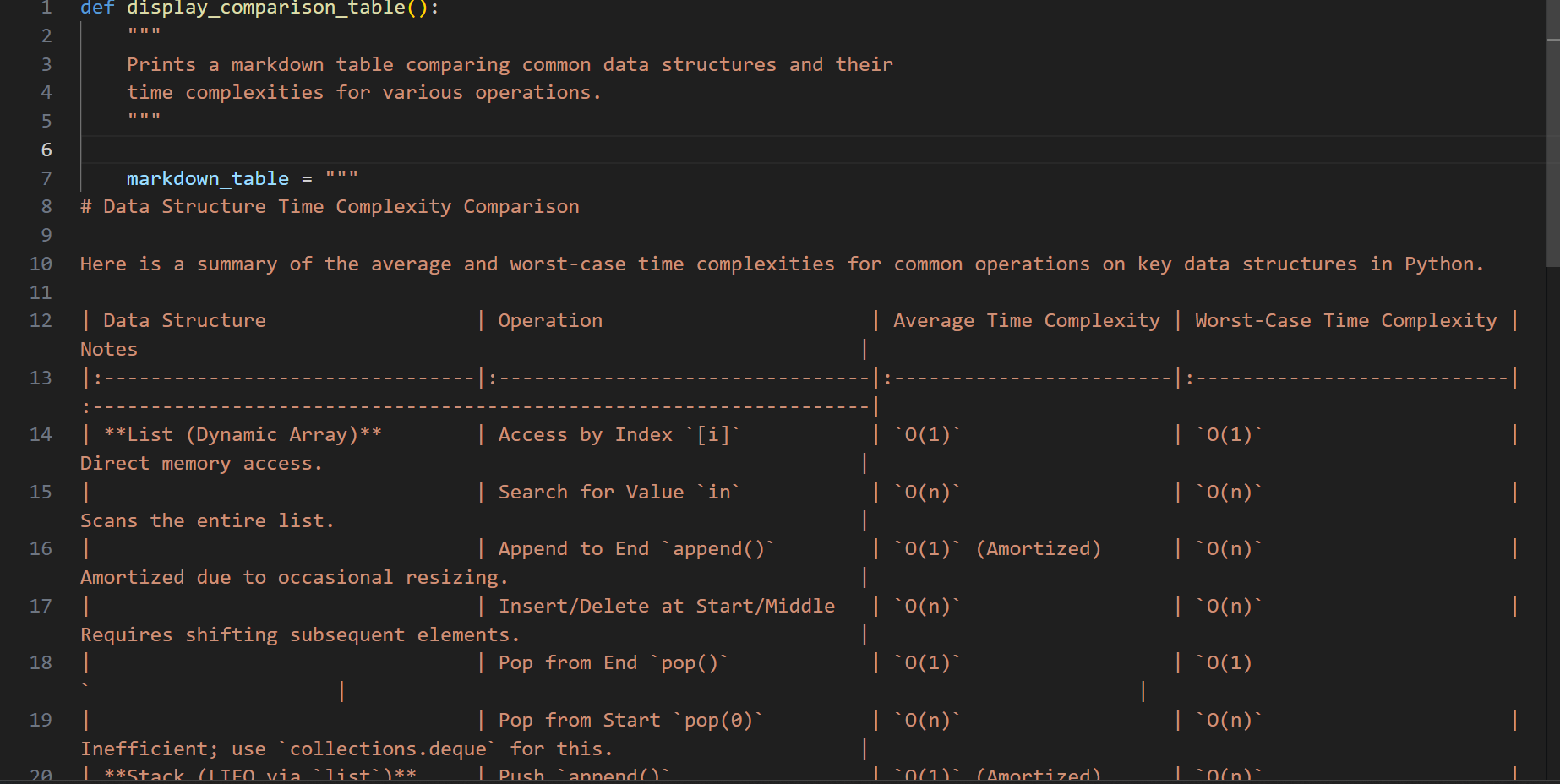
**QUESTION:**

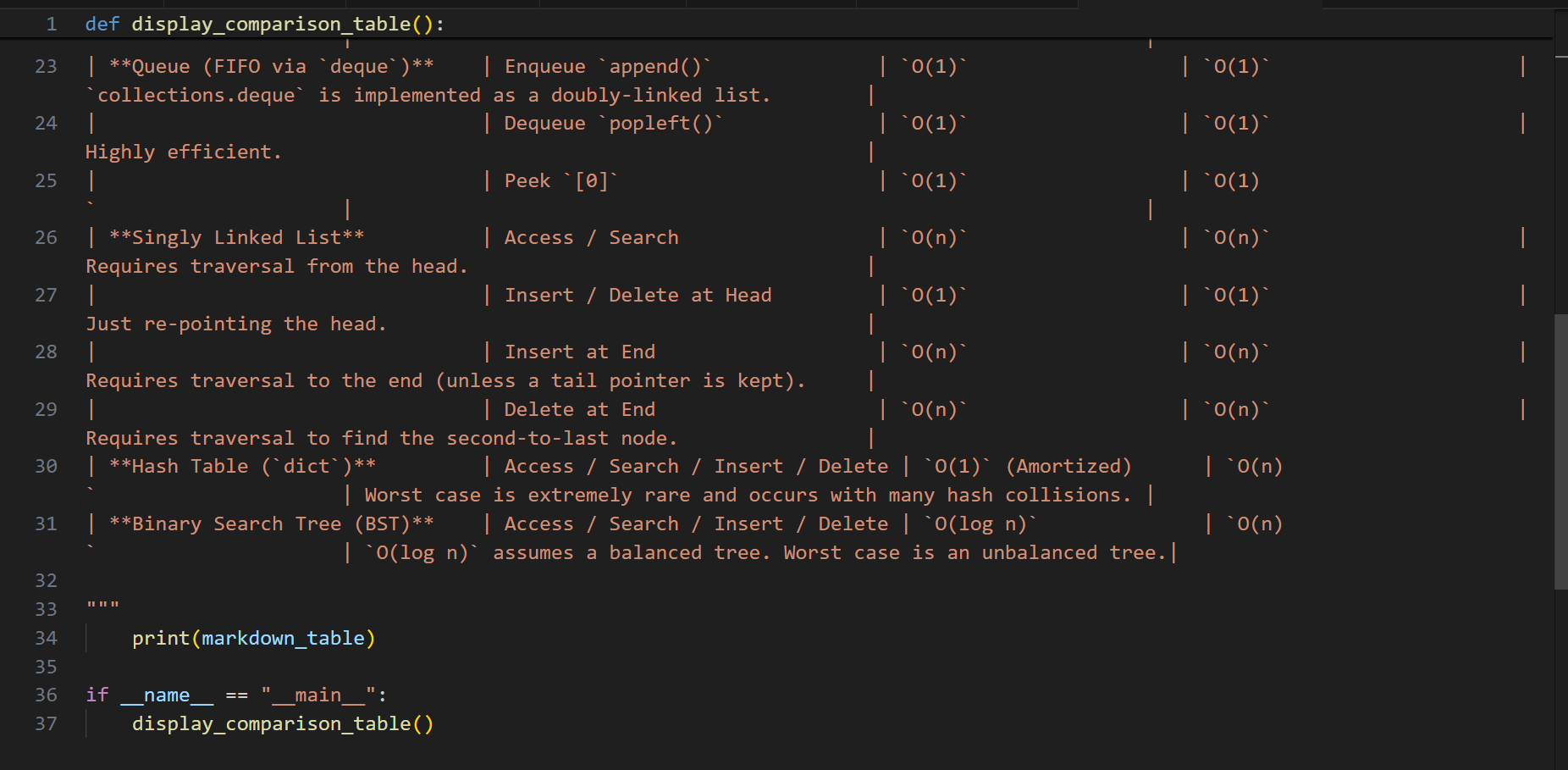
**9 – AI-Generated Data Structure Comparisons  
Task: Use AI to generate a comparison table of different data structures (stack,  
queue, linked list, etc.) including time complexities.  
Sample Input Code:  
# No code, prompt AI for a data structure comparison table  
Expected Output:  
• A markdown table with structure names, operations, and complexities**

**PROMPT:**

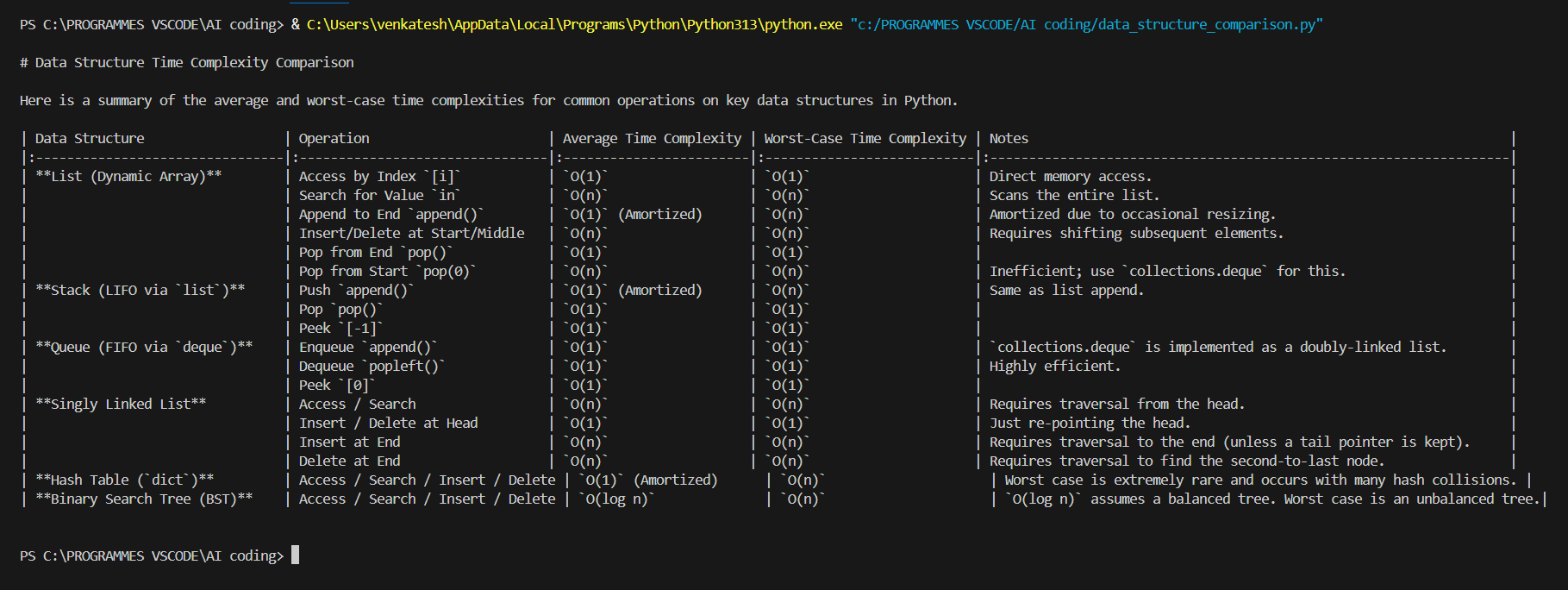
**Generate a python code by comparison table of different data structures (stack,  
queue, linked list, etc.) including time complexities.with a markdown table with structure names, operations, and complexities**

**CODE:**

****

****

**OUTPUT:**

****

**OBSERVATION:**

**This script provides a clear and concise summary of time complexities for common Python data structures by printing a pre-formatted markdown table. The information is accurate and well-organized, covering average and worst-case scenarios for essential operations like access, search, and insertion.**

## TASK-10

**QUESTION:**

**0 Real-Time Application Challenge – Choose the  
Right Data Structure  
Scenario:Your college wants to develop a Campus Resource Management System that  
handles:  
1. Student Attendance Tracking – Daily log of students entering/exiting  
the campus.  
2. Event Registration System – Manage participants in events with quick  
search and removal.  
3. Library Book Borrowing – Keep track of available books and their due  
dates.  
4. Bus Scheduling System – Maintain bus routes and stop connections.  
5. Cafeteria Order Queue – Serve students in the order they arrive.  
Student Task:  
• For each feature, select the most appropriate data structure from the list  
below:  
o Stack  
o Queue  
o Priority Queue  
o Linked List  
o Binary Search Tree (BST)  
o Graph  
o Hash Table  
o Deque  
• Justify your choice in 2–3 sentences per feature.  
• Implement one selected feature as a working Python program with AI-  
assisted code generation.  
Expected Output:  
• A table mapping feature → chosen data structure → justification.  
• A functional Python program implementing the chosen feature with  
comments and docstrings.  
 Deliverables (For All Tasks)  
1. AI-generated prompts for code and test case generation.  
2. At least 3 assert test cases for each task.  
3. AI-generated initial code and execution screenshots.  
4. Analysis of whether code passes all tests.  
5. Improved final version with inline comments and explanation.  
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code,  
and output**

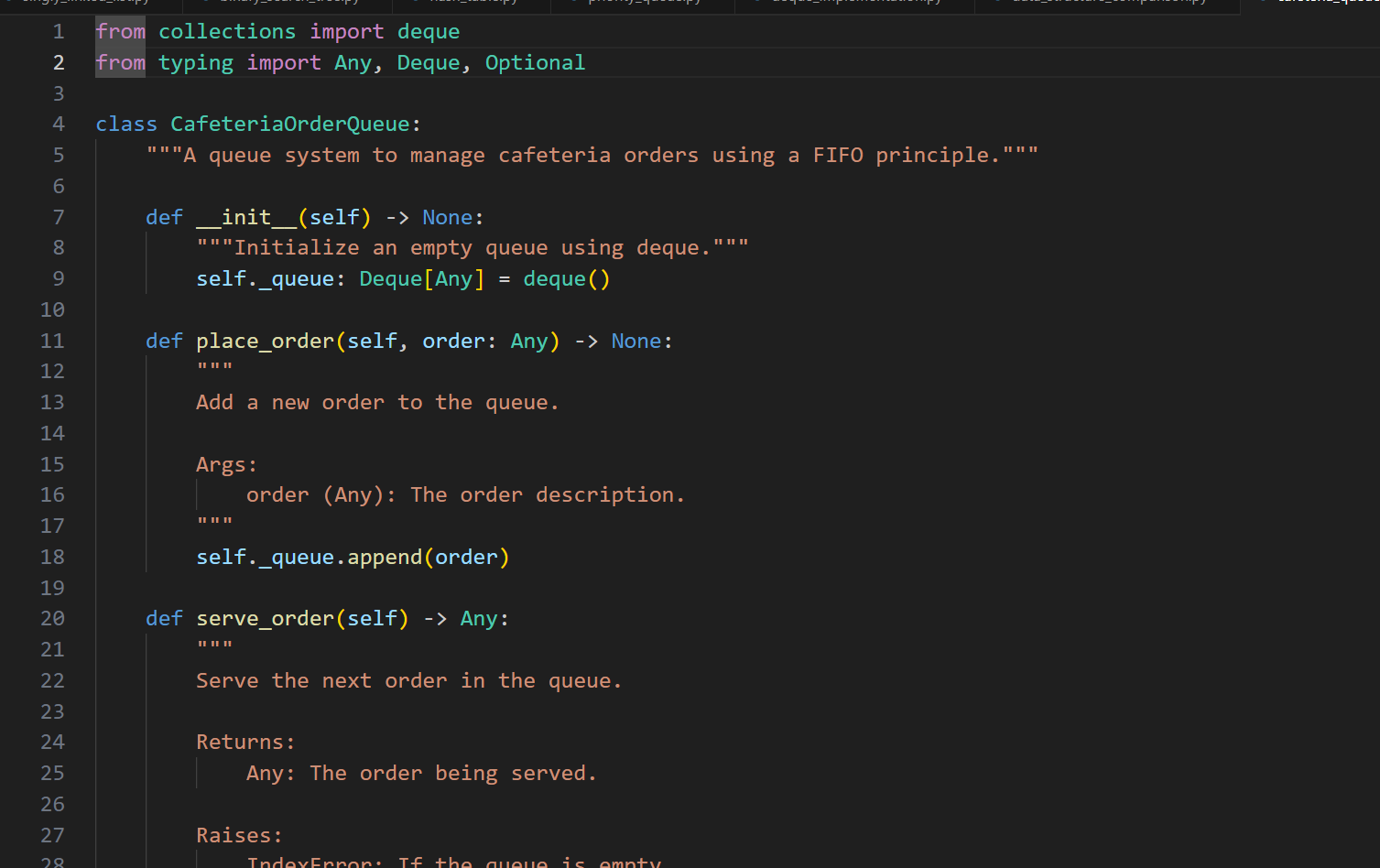
**PROMPT:**

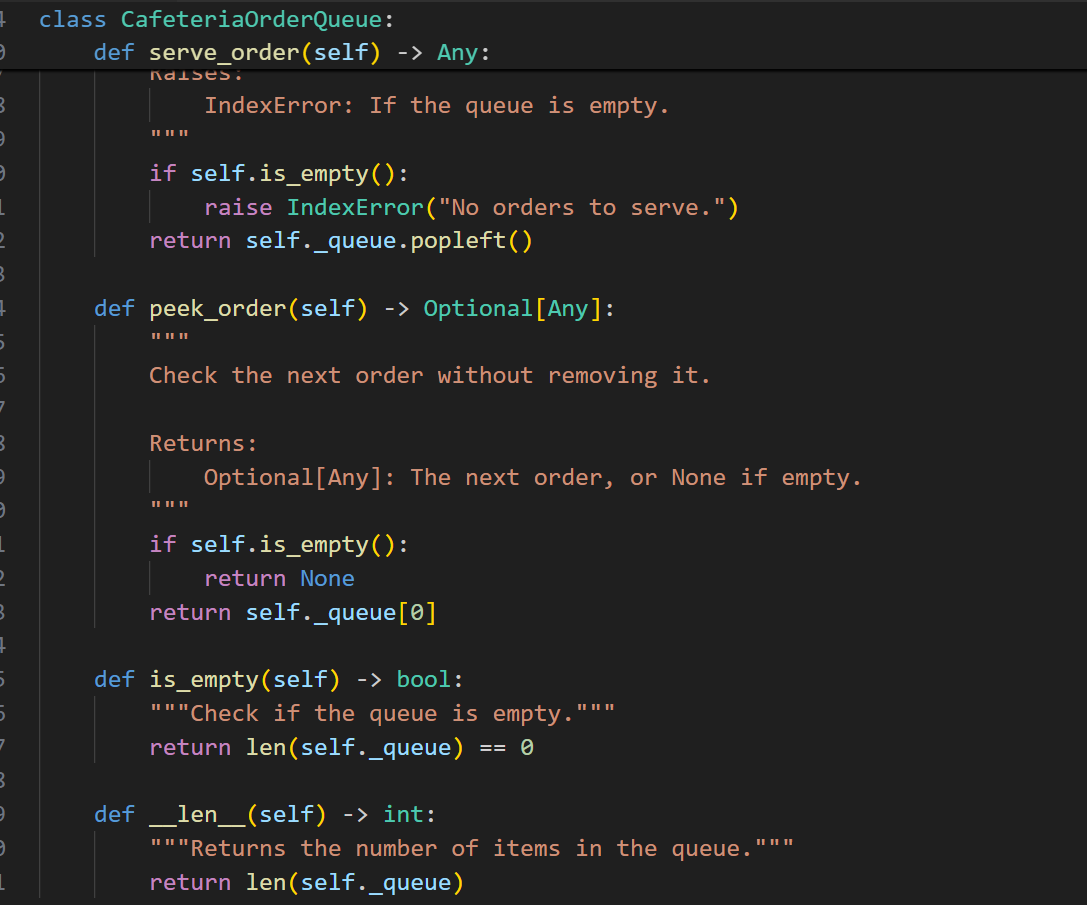
**Generate a python code For a Campus Resource Management System, map each feature to the best data structure from [Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, Deque]. Give justification in 2–3 sentences for each choice in a table format.**

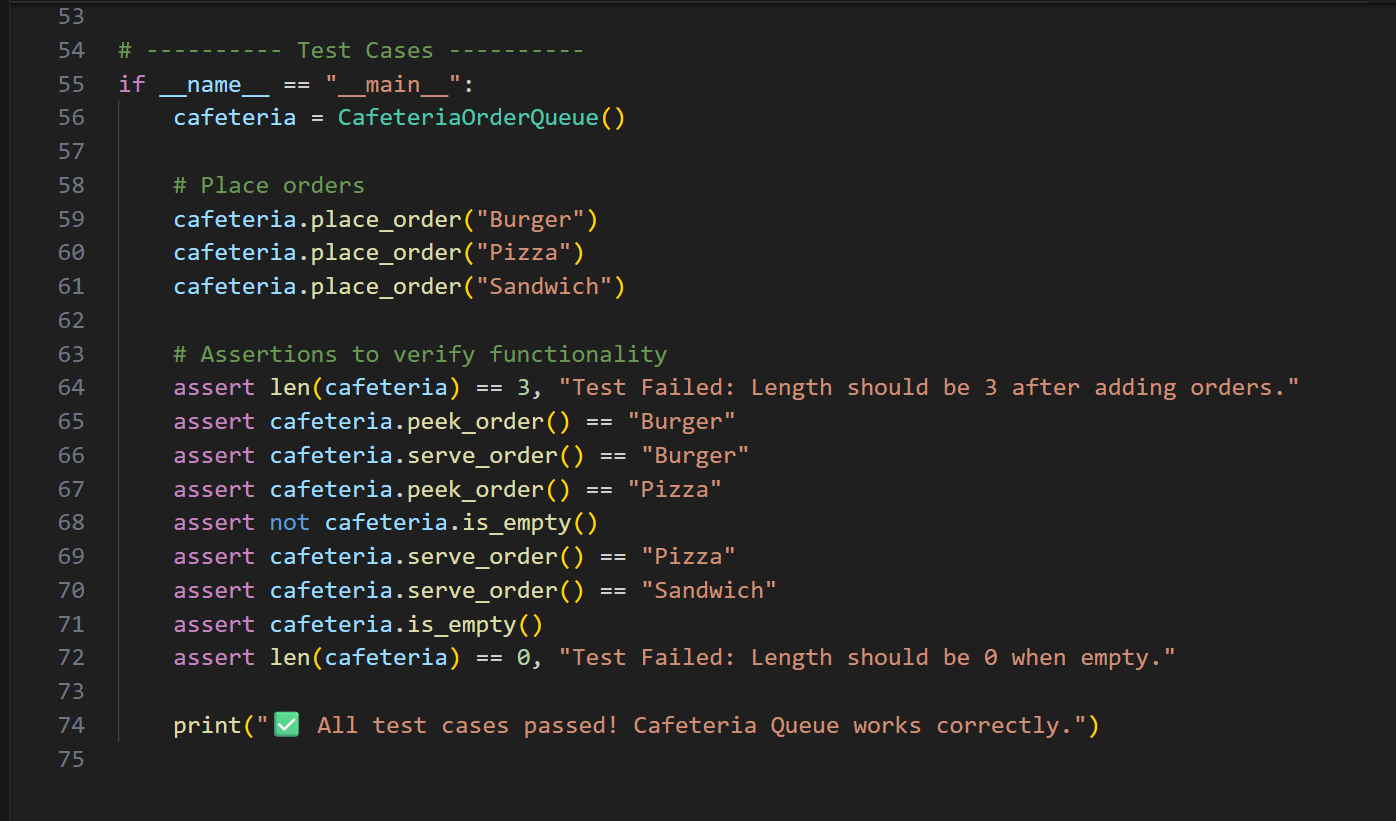
**Then, implement one feature (e.g., Cafeteria Order Queue using Queue) in Python**

**with docstrings, comments, and at least 3 assert test cases.**

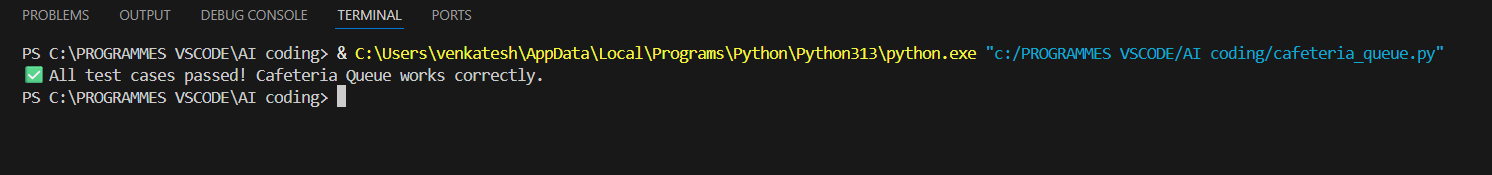
**CODE:**

****

****

****

**OUTPUT:**

****

**OBSERVATION:**

**This code provides a clean and efficient implementation of a FIFO queue, correctly leveraging collections deque for optimal O(1) performance on order operations. The class is made robust and reliable through clear method naming, comprehensive docstrings, and a solid suite of self-verifying assert test cases.**