Assignment-11.1

Data Structures with AI: Implementing Fundamental Structures

K. Santhosh Kumar

2403a51l21

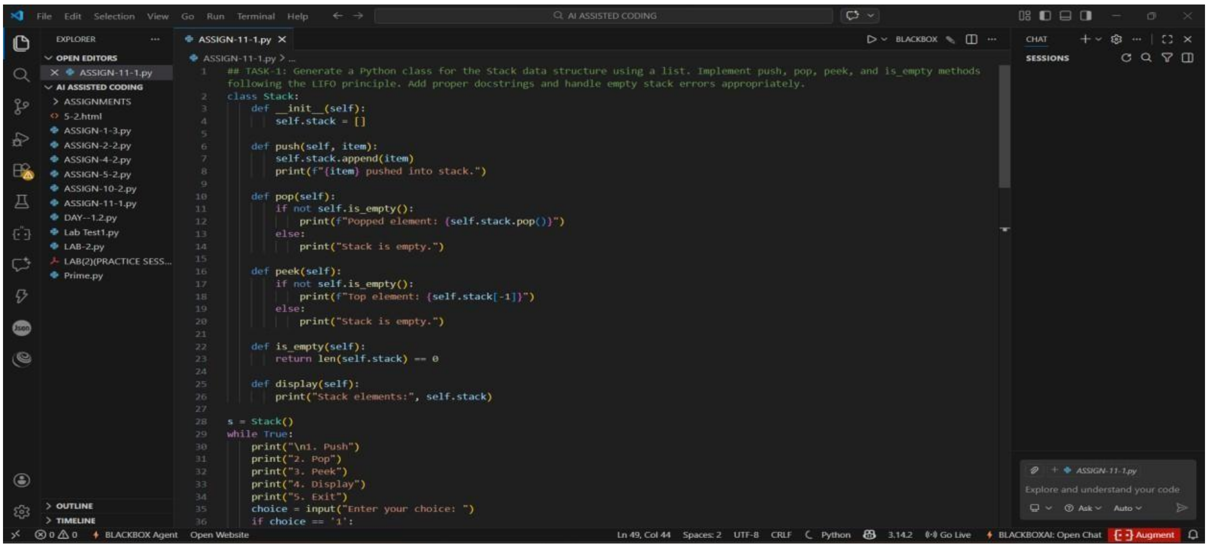
B-51

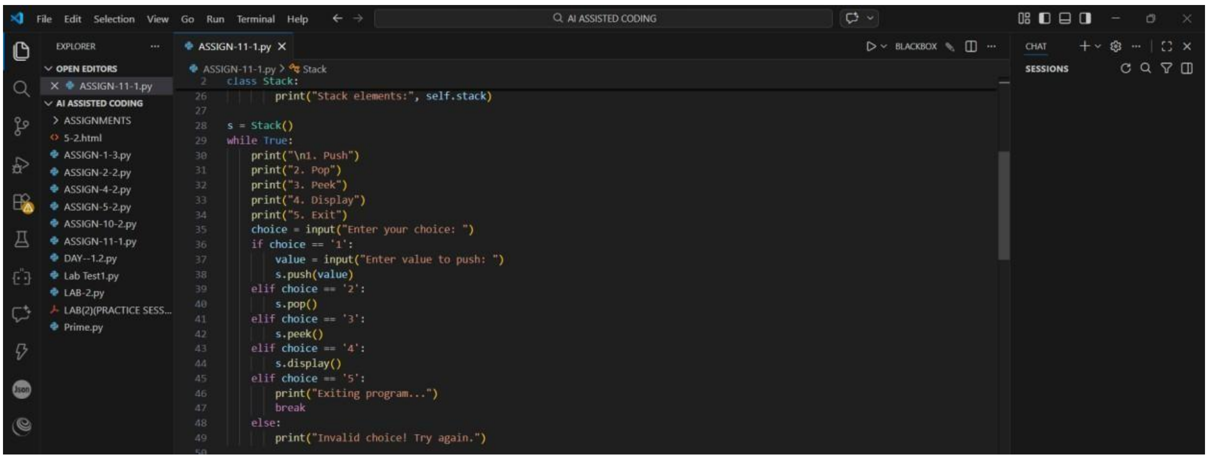
Task Description #1 - Stack Implementation:

Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.

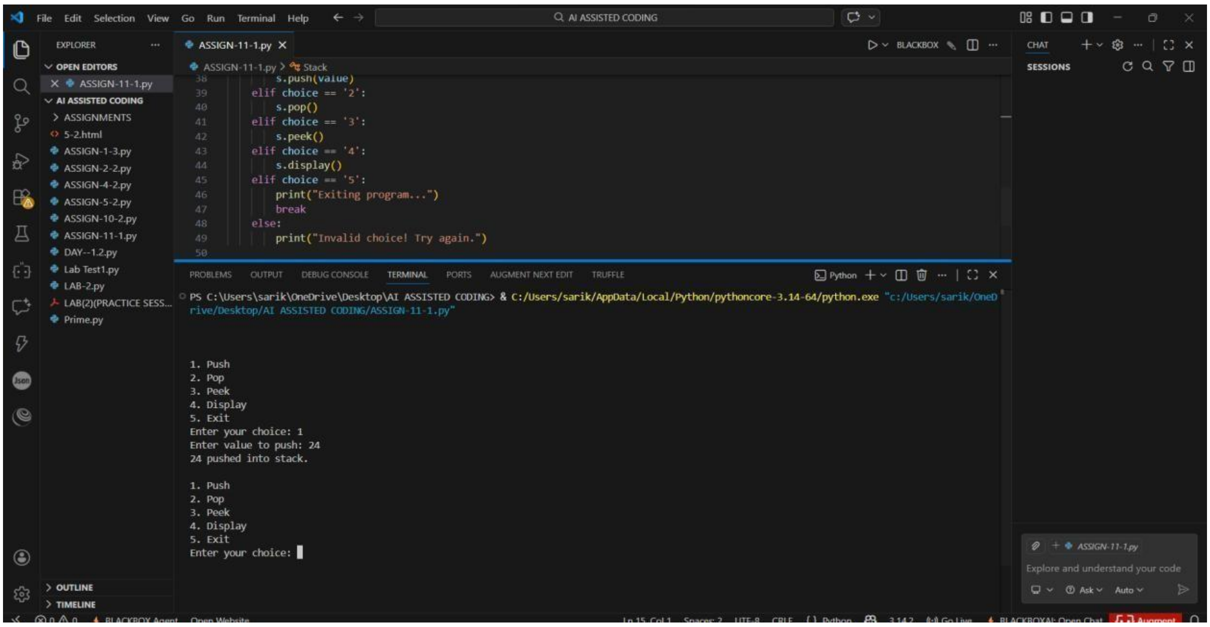
PROMPT: Generate a Python class for the Stack data structure using a list. Implement push, pop, peek, and is_empty methods following the LIFO principle. Add proper docstrings and handle empty stack errors appropriately.

Sample Input Code:





OUTPUT:



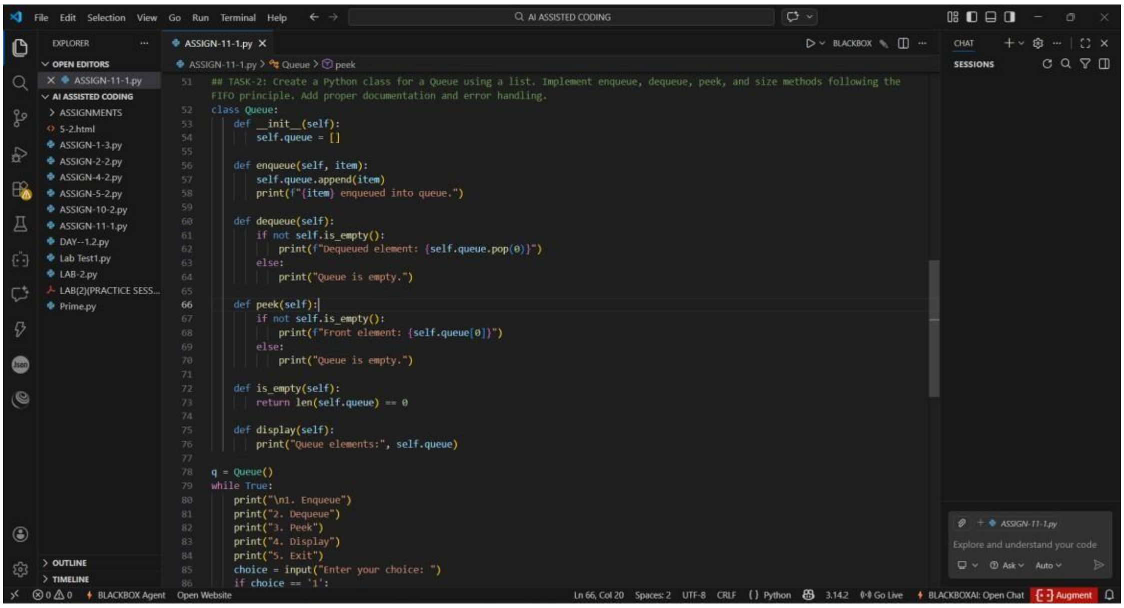
EXPLANATION: A Stack is a linear data structure that follows the LIFO (Last In First Out) principle, where the last element inserted is the first one removed. Operations such as push, pop, and peek are performed at one end called the top. It is commonly used in function calls, undo operations, and expression evaluation.

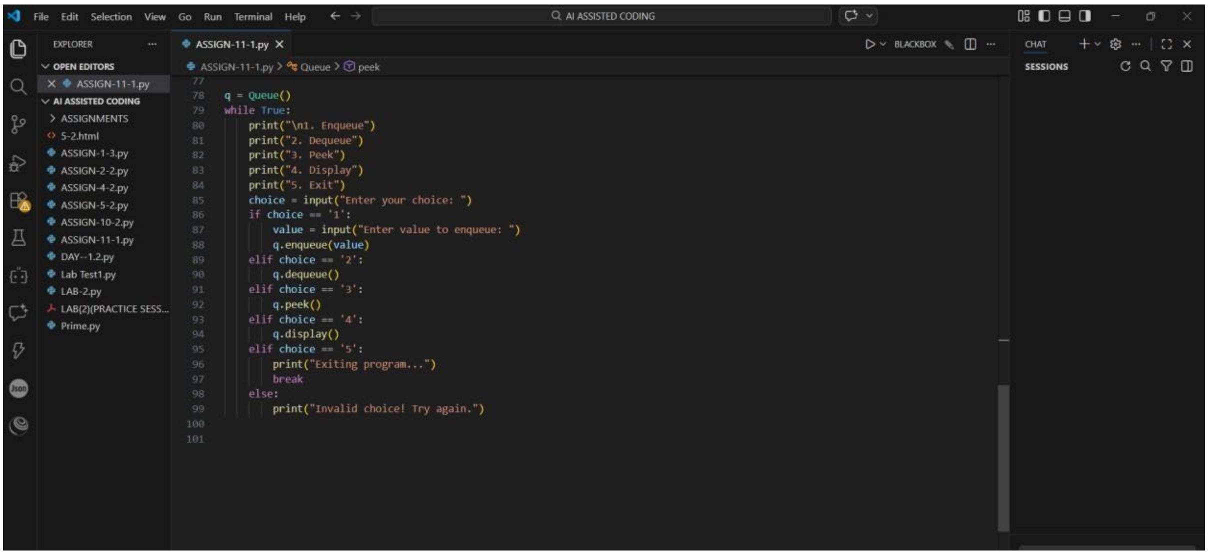
Task Description #2 – Queue Implementation:

Task: Use AI to implement a Queue using Python lists.

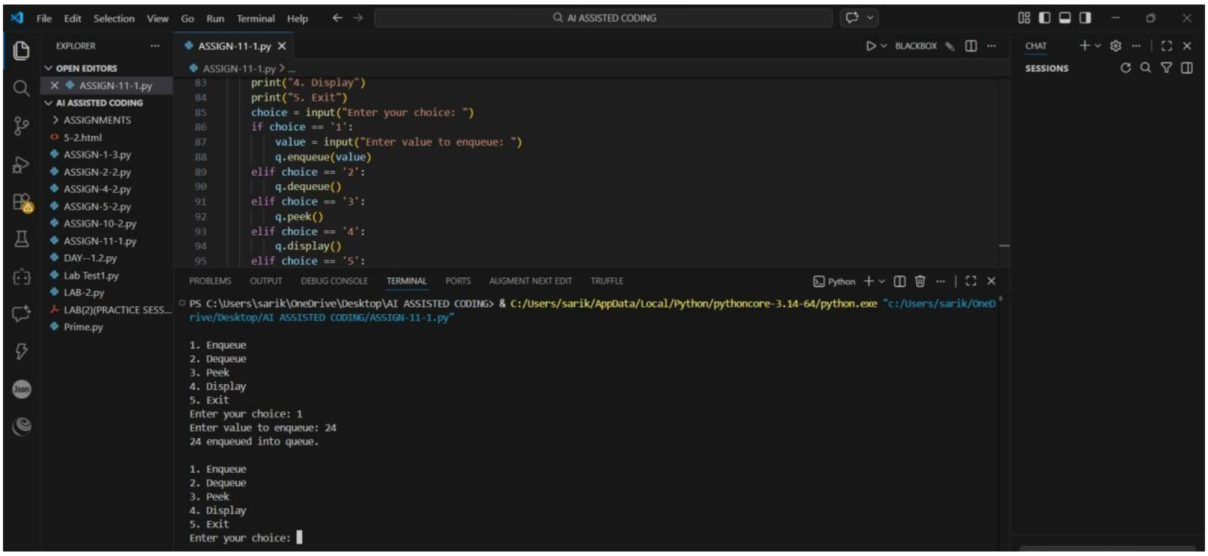
PROMPT: Create a Python class for a Queue using a list. Implement enqueue, dequeue, peek, and size methods following the FIFO principle. Add proper documentation and error handling.

Sample Input Code:





OUTPUT:



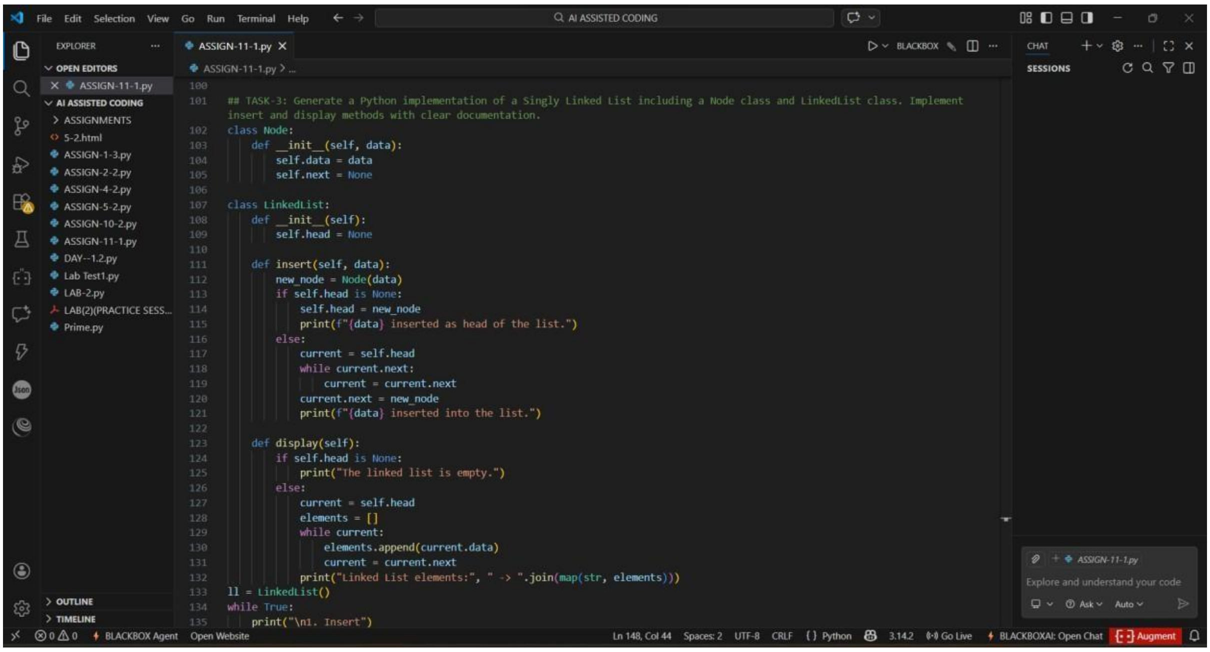
EXPLANATION: A Queue is a linear data structure that follows the FIFO (First In First Out) principle, where the first inserted element is removed first. Elements are added at the rear and removed from the front. It is widely used in scheduling systems, buffering, and real-world waiting line applications.

Task Description #3 – Linked List:

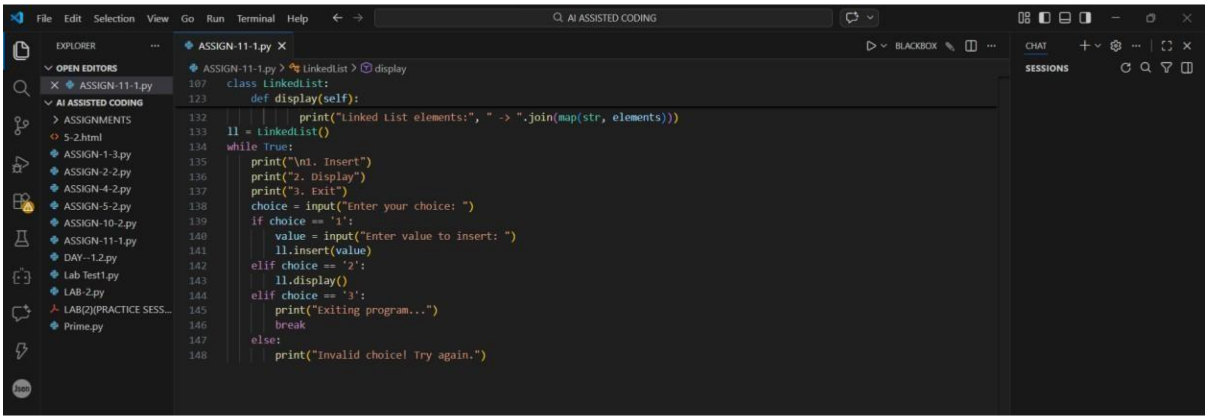
Task: Use AI to generate a Singly Linked List with insert and display methods.

PROMPT: Generate a Python implementation of a Singly Linked List including a Node class and LinkedList class. Implement insert and display methods with clear documentation.

Sample Input Code:

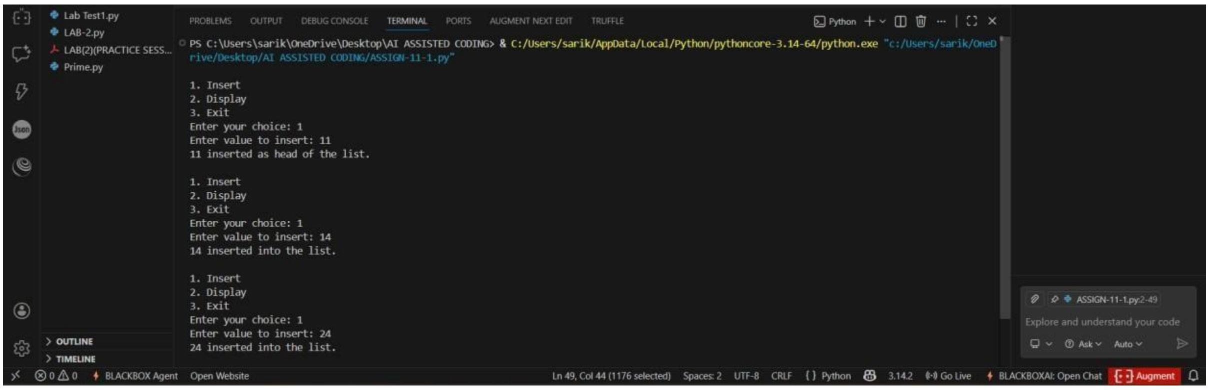


```python
## TASK-3: Generate a Python implementation of a Singly Linked List including a Node class and LinkedList class. Implement insert and display methods with clear documentation.
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            print(f"{data} inserted as head of the list.")
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
            print(f"{data} inserted into the list.")

    def display(self):
        if self.head is None:
            print("The linked list is empty.")
        else:
            current = self.head
            elements = []
            while current:
                elements.append(current.data)
                current = current.next
            print("Linked List elements:", " -> ".join(map(str, elements)))
ll = LinkedList()
while True:
    print("\n1. Insert")
```



```python
ll = LinkedList()
while True:
    print("\n1. Insert")
    print("2. Display")
    print("3. Exit")
    choice = input("Enter your choice: ")
    if choice == '1':
        value = input("Enter value to insert: ")
        ll.insert(value)
    elif choice == '2':
        ll.display()
    elif choice == '3':
        print("Exiting program...")
        break
    else:
        print("Invalid choice! Try again.")
```

OUTPUT:



```
PS C:\Users\sarik\OneDrive\Desktop\AI ASSISTED CODING> & C:/Users/sarik/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/sarik/OneDrive/Desktop/AI ASSISTED CODING/ASSIGN-11-1.py"

1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter value to insert: 11
11 inserted as head of the list.

1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter value to insert: 14
14 inserted into the list.

1. Insert
2. Display
3. Exit
Enter your choice: 1
Enter value to insert: 24
24 inserted into the list.
```

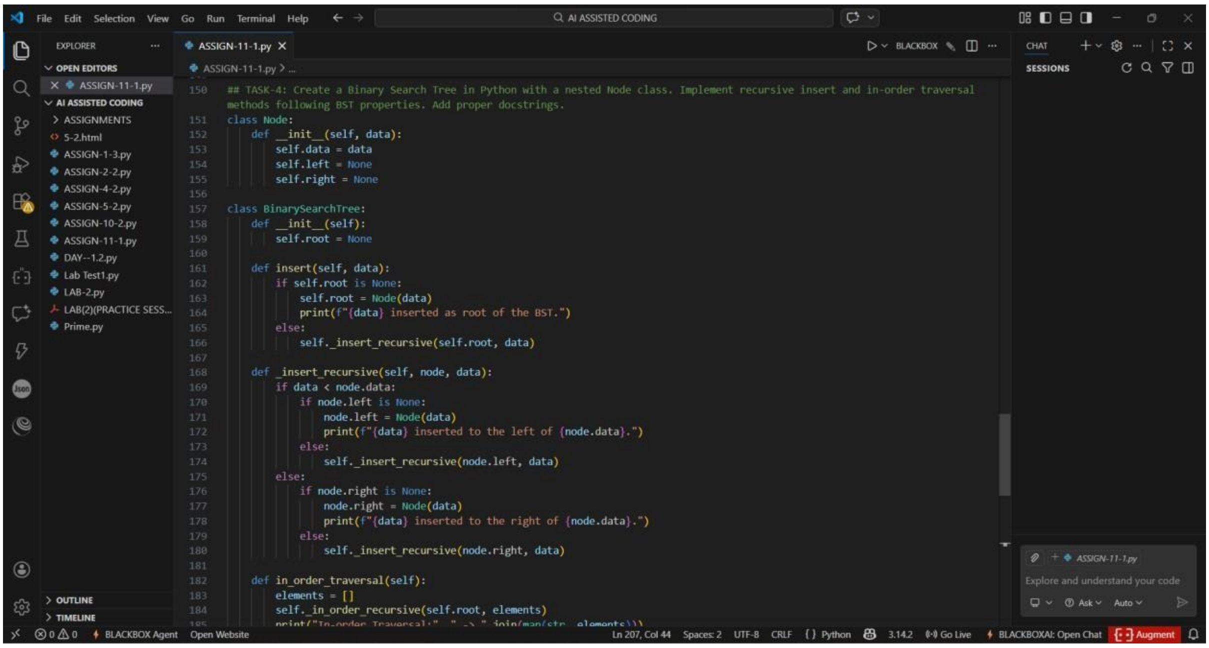
EXPLANATION: A Singly Linked List consists of nodes where each node contains data and a reference to the next node. Unlike arrays, it does not require contiguous memory, making it dynamic in size. It allows efficient insertions and deletions compared to fixed- size structures.

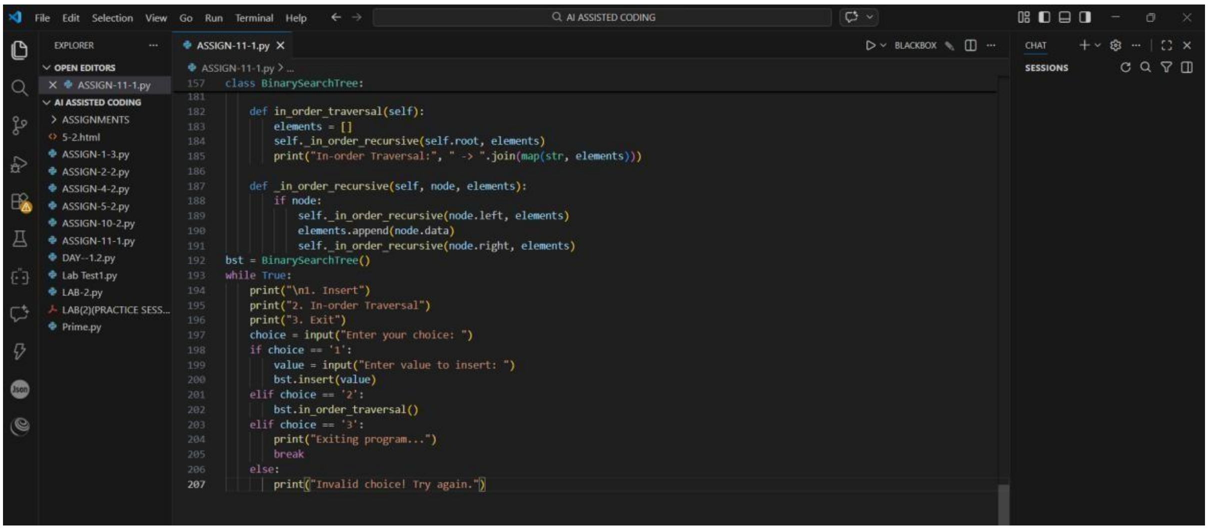
Task Description #4 – Binary Search Tree (BST):

Task: Use AI to create a BST with insert and in-order traversal methods.

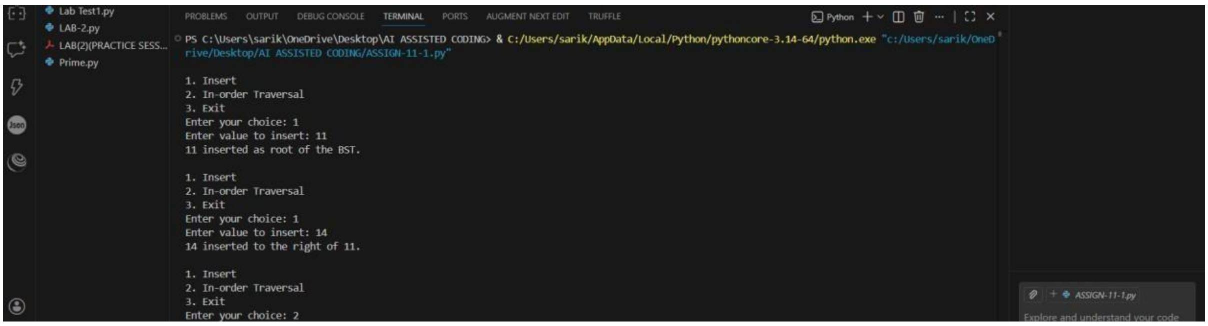
PROMPT: Create a Binary Search Tree in Python with a nested Node class. Implement recursive insert and in-order traversal methods following BST properties. Add proper docstrings.

Sample Input Code:





OUTPUT:



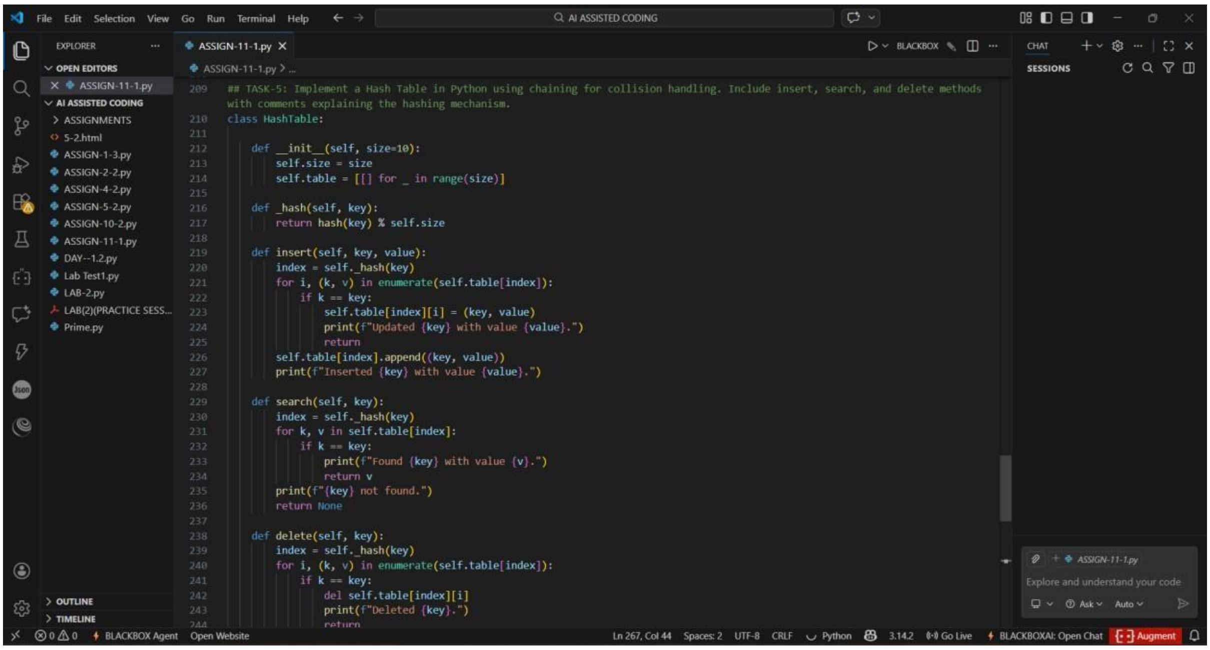
EXPLANATION: A Binary Search Tree is a hierarchical data structure where the left child contains smaller values and the right child contains larger values than the root. This property makes searching, insertion, and deletion efficient. In-order traversal of a BST produces sorted output.

Task Description #5 – Hash Table:

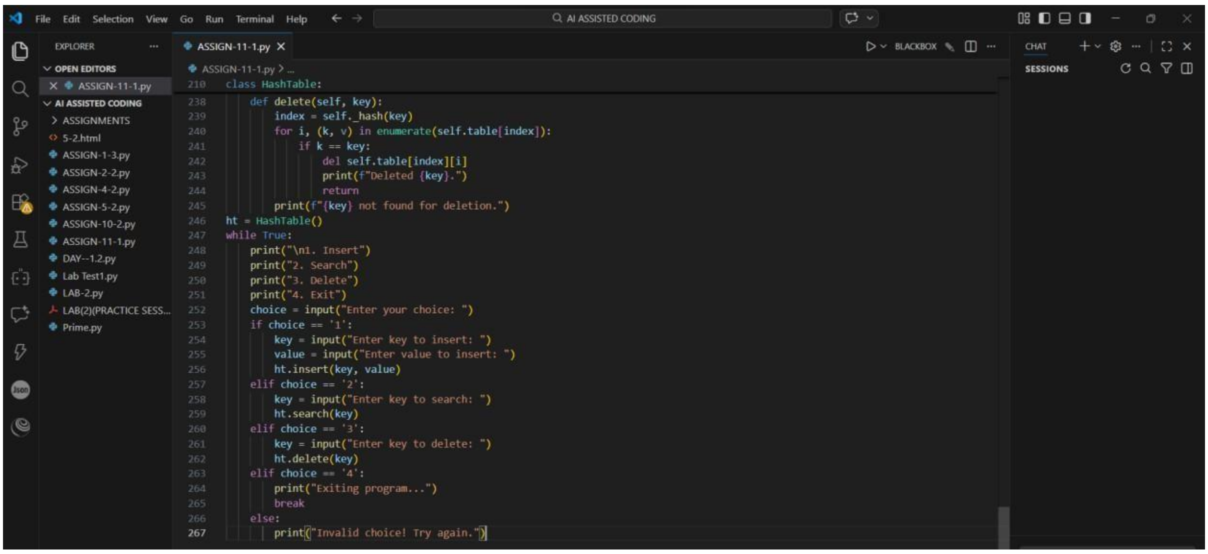
Task: Use AI to implement a hash table with basic insert, search, and delete methods.

PROMPT: Implement a Hash Table in Python using chaining for collision handling. Include insert, search, and delete methods with comments explaining the hashing mechanism.

Sample Input Code:

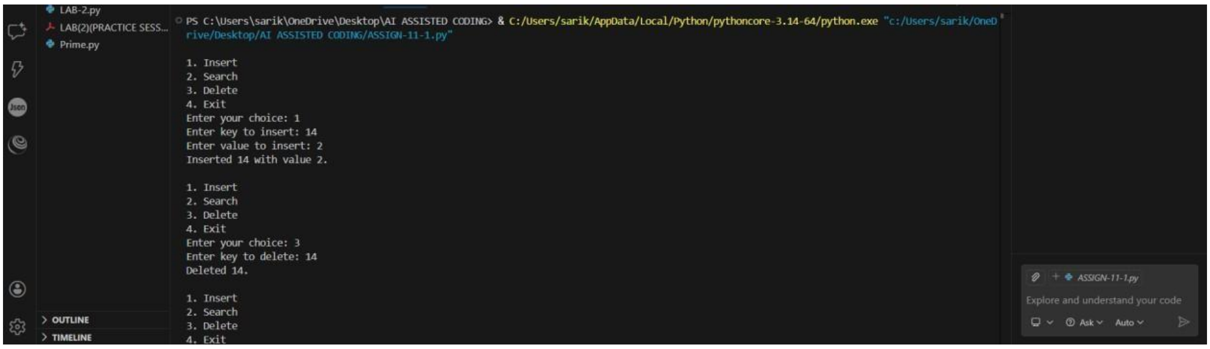


```python
## TASK-5: Implement a Hash Table in Python using chaining for collision handling. Include insert, search, and delete methods
with comments explaining the hashing mechanism.
class HashTable:

    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                print(f"Updated {key} with value {value}.")
                return
        self.table[index].append((key, value))
        print(f"Inserted {key} with value {value}.")

    def search(self, key):
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                print(f"Found {key} with value {v}.")
                return v
        print(f"{key} not found.")
        return None

    def delete(self, key):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                print(f"Deleted {key}.")
                return
```



```python
class HashTable:
    def delete(self, key):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                print(f"Deleted {key}.")
                return
        print(f"{key} not found for deletion.")
ht = HashTable()
while True:
    print("\n1. Insert")
    print("2. Search")
    print("3. Delete")
    print("4. Exit")
    choice = input("Enter your choice: ")
    if choice == '1':
        key = input("Enter key to insert: ")
        value = input("Enter value to insert: ")
        ht.insert(key, value)
    elif choice == '2':
        key = input("Enter key to search: ")
        ht.search(key)
    elif choice == '3':
        key = input("Enter key to delete: ")
        ht.delete(key)
    elif choice == '4':
        print("Exiting program...")
        break
    else:
        print("Invalid choice! Try again.")
```

OUTPUT:



EXPLANATION: A Hash Table stores data in key-value pairs using a hash function to compute an index. It provides fast average-case time complexity for search, insertion, and deletion operations. Collisions are handled using techniques such as chaining.

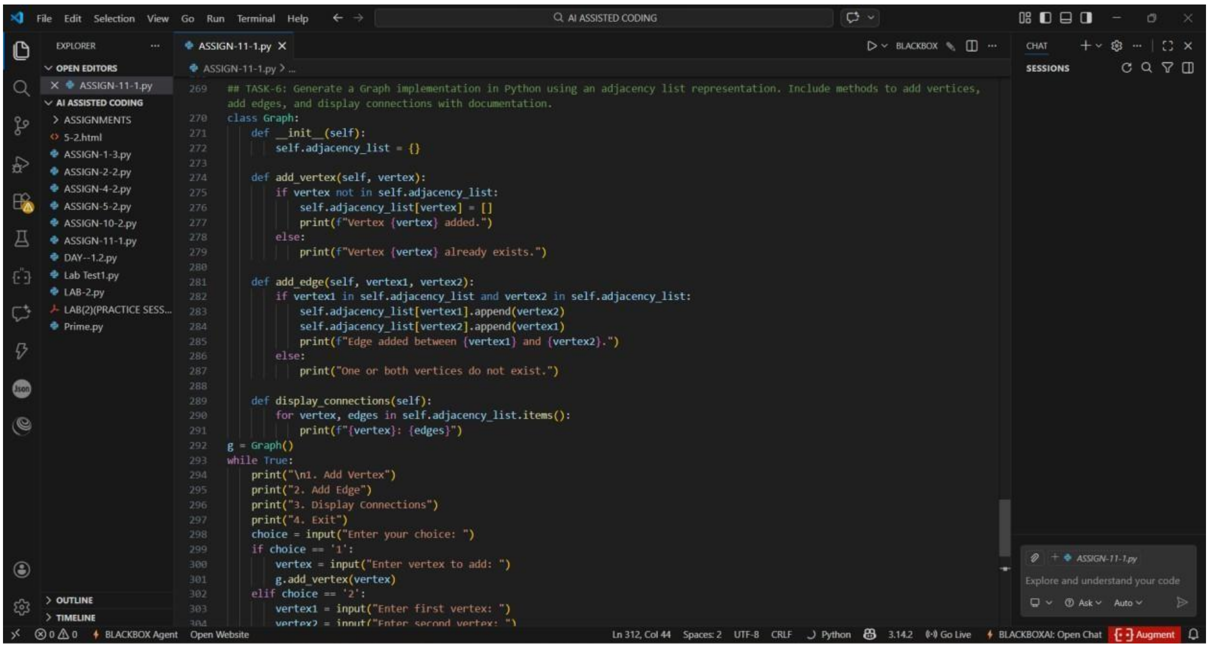
Task Description #6 – Graph Representation:

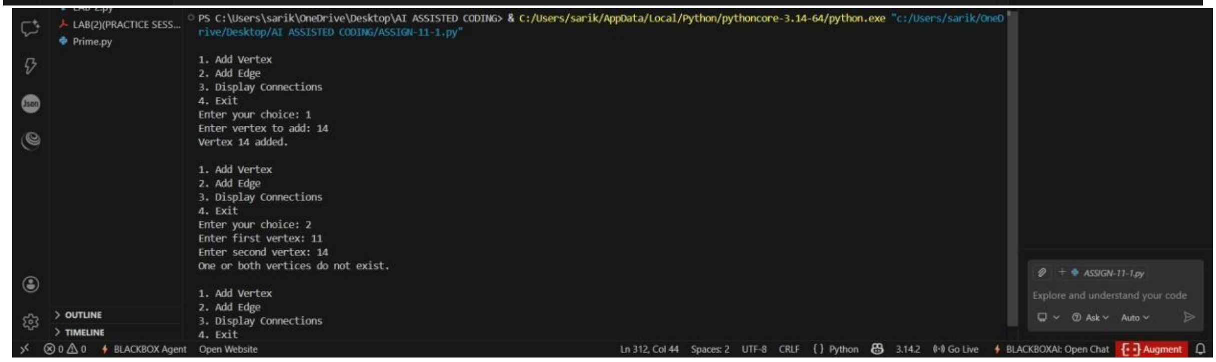
Task: Use AI to implement a graph using an adjacency list.

PROMPT: Generate a Graph implementation in Python using an adjacency list representation. Include methods to add vertices, add edges, and display connections with documentation.

Sample Input Code:

OUTPUT:





EXPLANATION: A Graph is a non-linear data structure used to represent relationships between entities. It consists of vertices (nodes) and edges (connections). Graphs are commonly used in networks, maps, and routing systems.

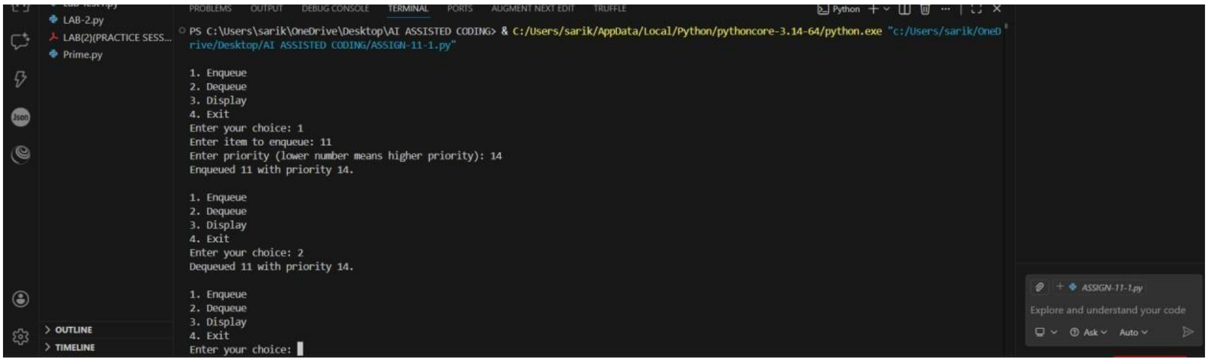
Task Description #7 – Priority Queue:

OUTPUT:

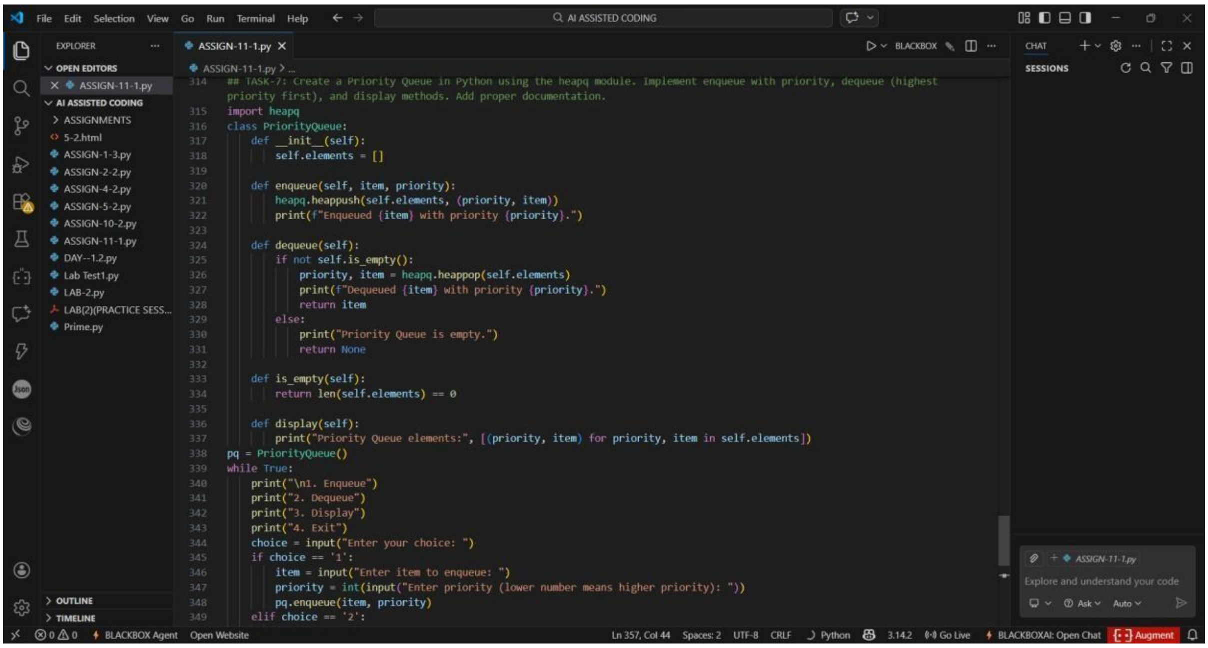
Task: Use AI to implement a priority queue using Python's heapq module.

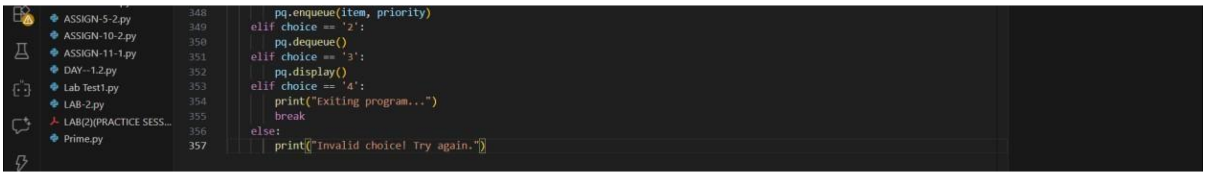
PROMPT: Create a Priority Queue in Python using the heapq module. Implement enqueue with priority, dequeue (highest priority first), and display methods. Add proper documentation.

Sample Input Code:



EXPLANATION: A Priority Queue is a special type of queue where elements are removed based on priority rather than order of insertion. Higher priority elements are processed first. It is typically implemented using a heap for efficiency.





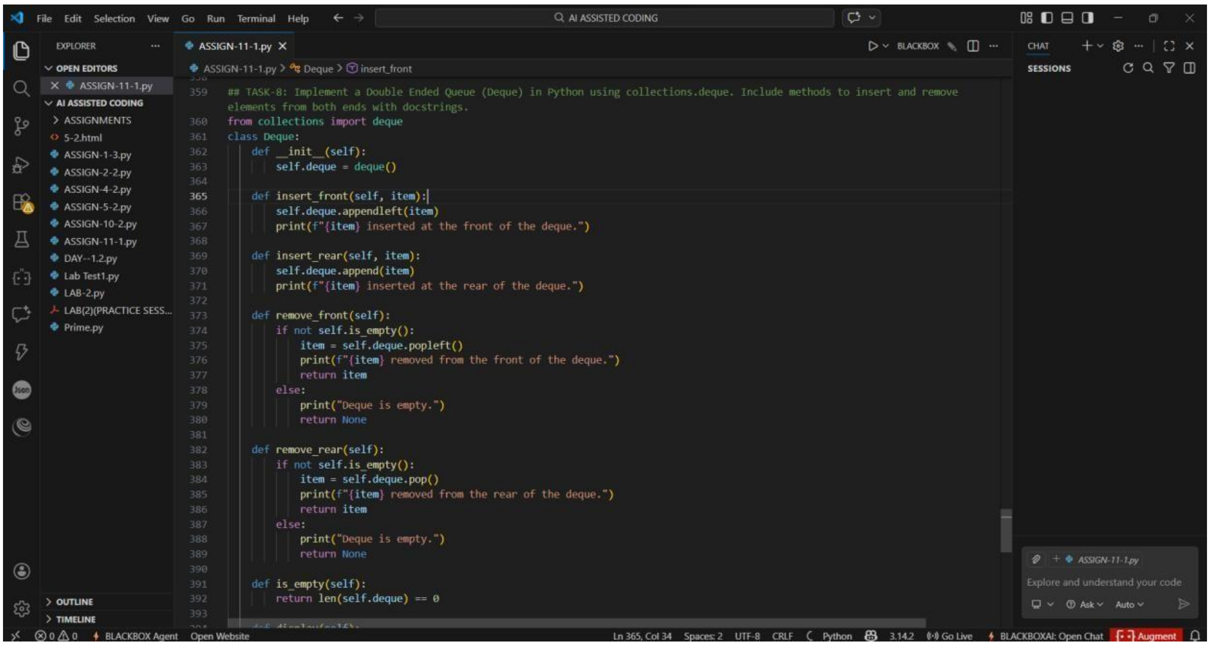
OUTPUT:

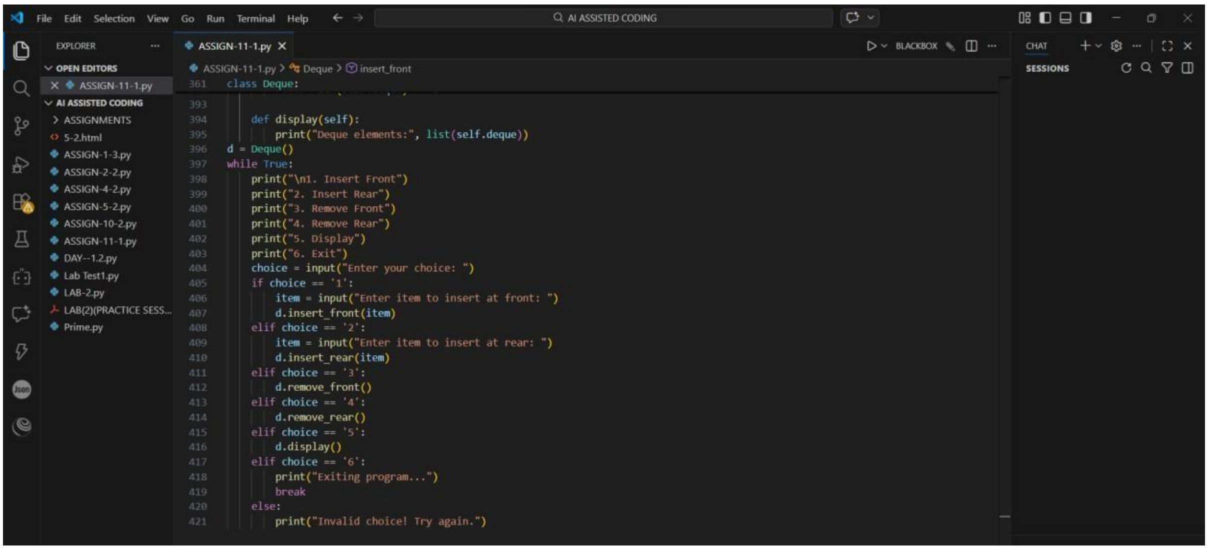
Task Description #8 – Deque:

Task: Use AI to implement a double-ended queue using Collections.deque.

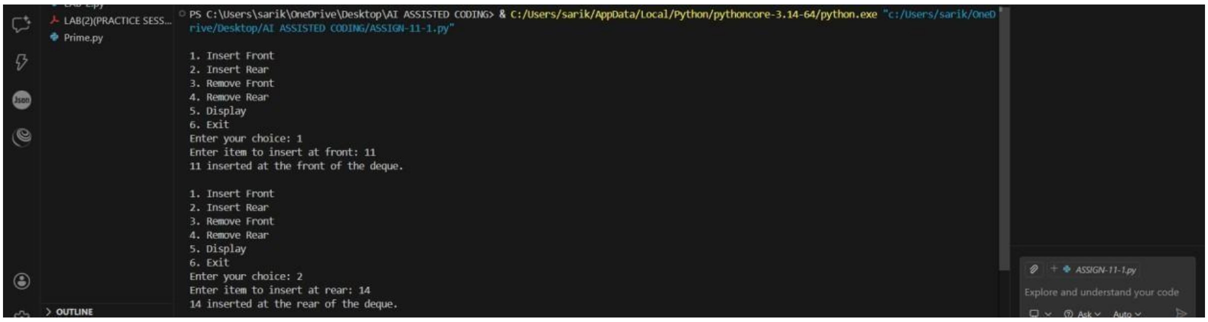
PROMPT: Implement a Double Ended Queue (Deque) in Python using collections.deque. Include methods to insert and remove elements from both ends with docstrings.

Sample Input Code:





OUTPUT:



EXPLANATION: A Deque (Double Ended Queue) allows insertion and deletion of elements from both the front and rear ends. It combines features of both stacks and queues. It is useful in applications like sliding window algorithms and task scheduling.