ASSIGNMENT-11.1

HT NO:2303A51905

Batch No:28

Course :AI Assisted Coding

Task Description 1:Stack Implementation

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

methods.

Sample Input Code:

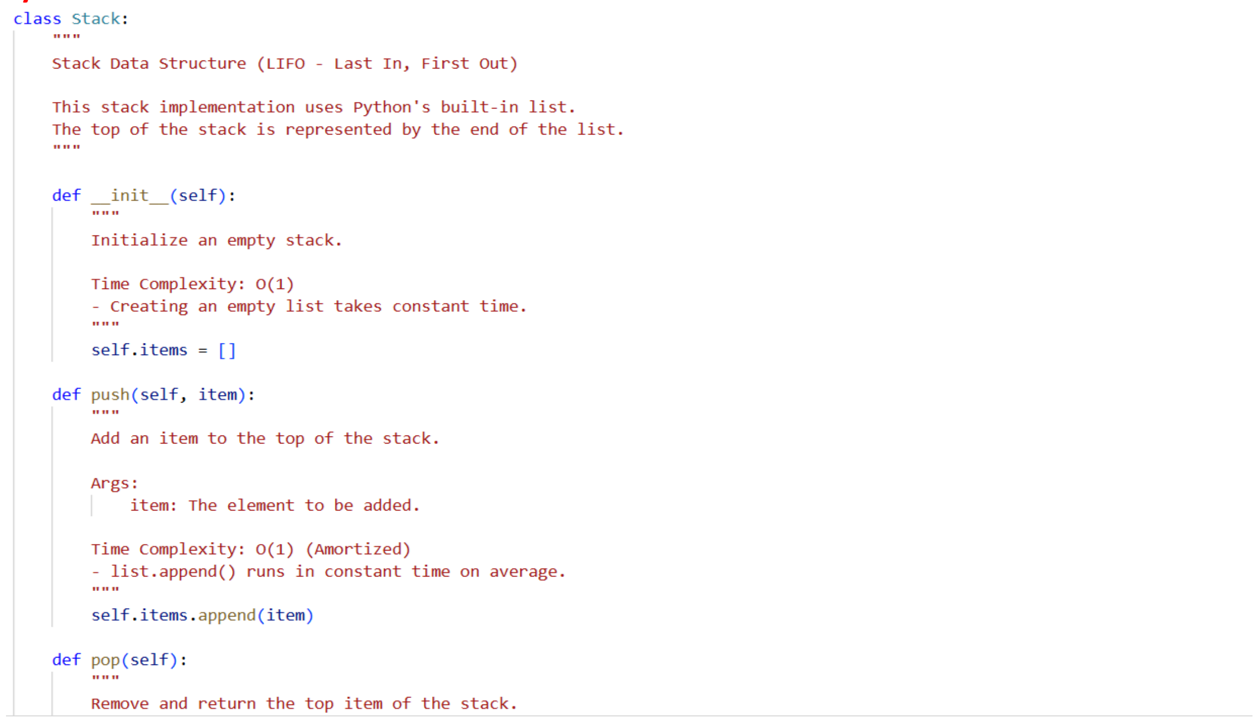
class Stack:

pass

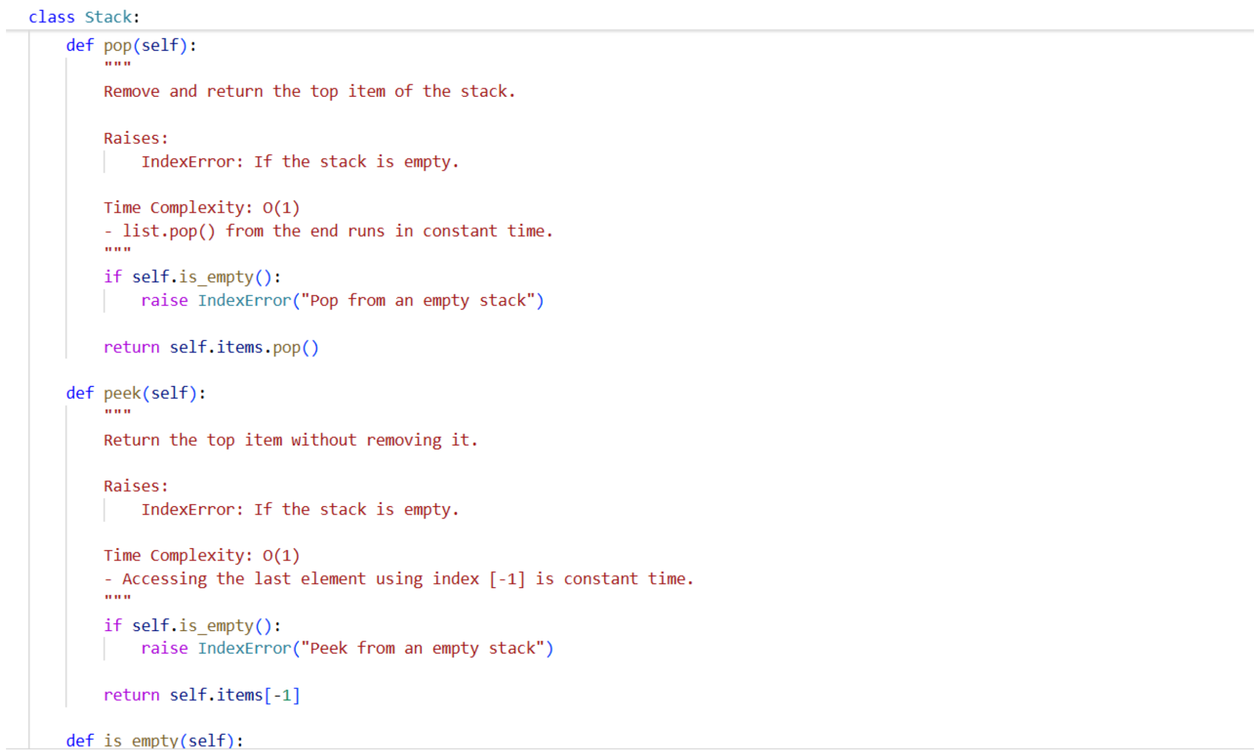
Prompt:

Improve the Stack implementation by adding meaningful comments and explaining the time complexity of each method.

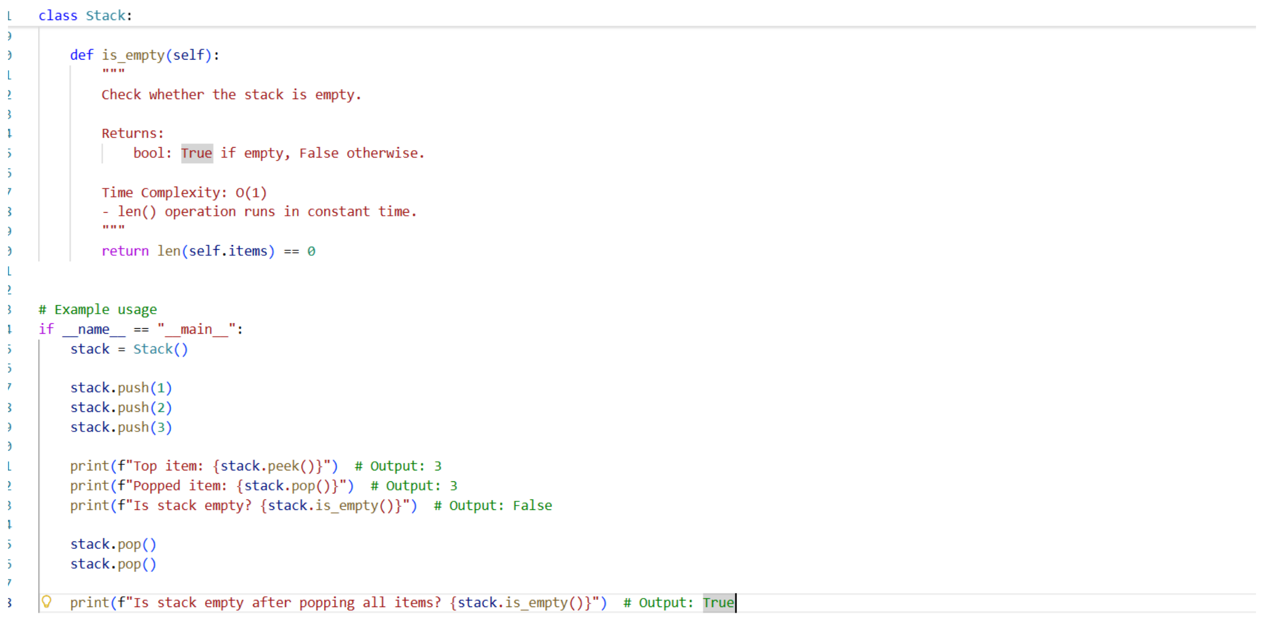
Python code:



```
class Stack:
    """
    Stack Data Structure (LIFO - Last In, First Out)

    This stack implementation uses Python's built-in list.
    The top of the stack is represented by the end of the list.
    """

    def __init__(self):
        """
        Initialize an empty stack.

        Time Complexity: O(1)
        - Creating an empty list takes constant time.
        """
        self.items = []

    def push(self, item):
        """
        Add an item to the top of the stack.

        Args:
            item: The element to be added.

        Time Complexity: O(1) (Amortized)
        - list.append() runs in constant time on average.
        """
        self.items.append(item)

    def pop(self):
        """
        Remove and return the top item of the stack.
```

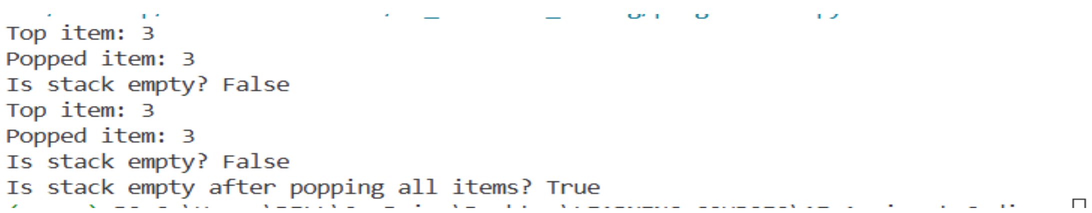


```python
class Stack:
    def pop(self):
        """
        Remove and return the top item of the stack.

        Raises:
            IndexError: If the stack is empty.

        Time Complexity: O(1)
        - list.pop() from the end runs in constant time.
        """
        if self.is_empty():
            raise IndexError("Pop from an empty stack")

        return self.items.pop()

    def peek(self):
        """
        Return the top item without removing it.

        Raises:
            IndexError: If the stack is empty.

        Time Complexity: O(1)
        - Accessing the last element using index [-1] is constant time.
        """
        if self.is_empty():
            raise IndexError("Peek from an empty stack")

        return self.items[-1]

    def is_empty(self):
```



```python
class Stack:

    def is_empty(self):
        """
        Check whether the stack is empty.

        Returns:
            bool: True if empty, False otherwise.

        Time Complexity: O(1)
        - len() operation runs in constant time.
        """
        return len(self.items) == 0


# Example usage
if __name__ == "__main__":
    stack = Stack()

    stack.push(1)
    stack.push(2)
    stack.push(3)

    print(f"Top item: {stack.peek()}")  # Output: 3
    print(f"Popped item: {stack.pop()}")  # Output: 3
    print(f"Is stack empty? {stack.is_empty()}")  # Output: False

    stack.pop()
    stack.pop()

    print(f"Is stack empty after popping all items? {stack.is_empty()}")  # Output: True
```

Output:



```
Top item: 3
Popped item: 3
Is stack empty? False
Top item: 3
Popped item: 3
Is stack empty? False
Is stack empty after popping all items? True
```

Justification:

1.A Stack follows the LIFO principle — the last element added is removed first.

2.push() adds an element to the top of the stack.

3.pop() removes and returns the top element.

4.peek() returns the top element without removing it.

5.is_empty() checks whether the stack contains any elements.

Task Description 2: Queue Implementation

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

Sample Input Code:

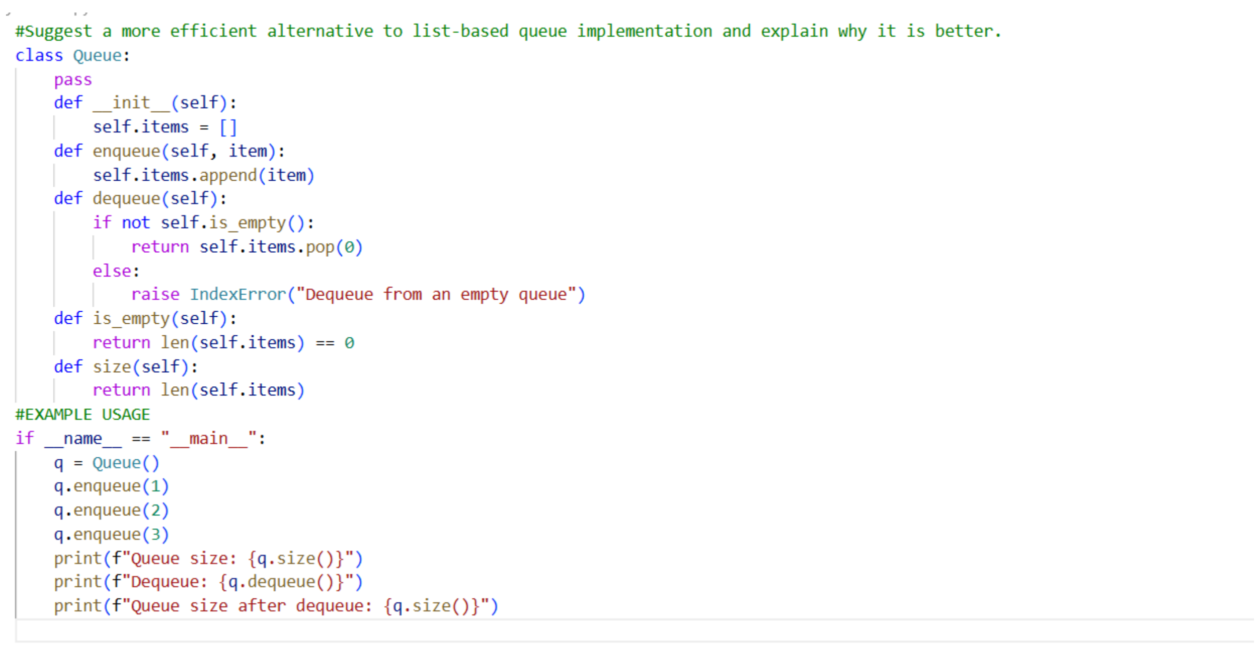
class Queue:

pass

Prompt:

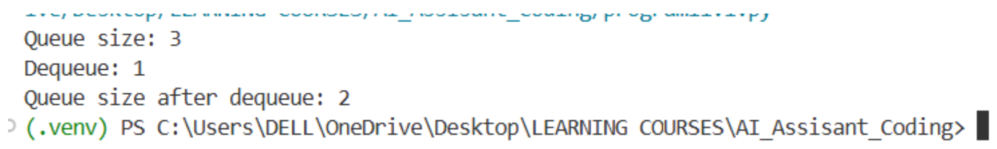
Suggest a more efficient alternative to list-based queue implementation and explain why it is better.

Python Code:



```python
#Suggest a more efficient alternative to list-based queue implementation and explain why it is better.
class Queue:
    pass
    def __init__(self):
        self.items = []
    def enqueue(self, item):
        self.items.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            raise IndexError("Dequeue from an empty queue")
    def is_empty(self):
        return len(self.items) == 0
    def size(self):
        return len(self.items)
#EXAMPLE USAGE
if __name__ == "__main__":
    q = Queue()
    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)
    print(f"Queue size: {q.size()}")
    print(f"Dequeue: {q.dequeue()}")
    print(f"Queue size after dequeue: {q.size()}")
```

Output:



```
Queue size: 3
Dequeue: 1
Queue size after dequeue: 2
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
```

Justification:

1.Queue follows FIFO order — first inserted element is removed first.

2.enqueue() adds elements to the rear of the queue.

3.dequeue() removes elements from the front.

4.peek() returns the front element without removing it.

5.size() returns the total number of elements in the queue.

Task Description 3 : Linked List

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

methods.

Sample Input Code:

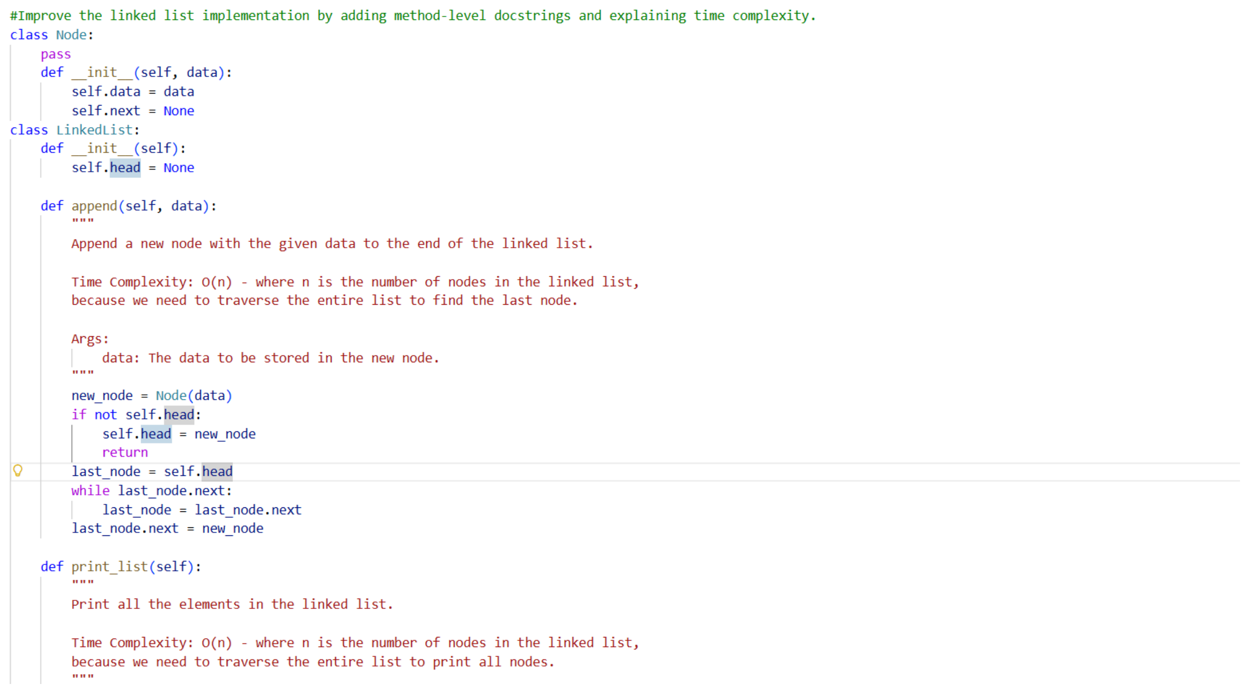
class Node:

pass

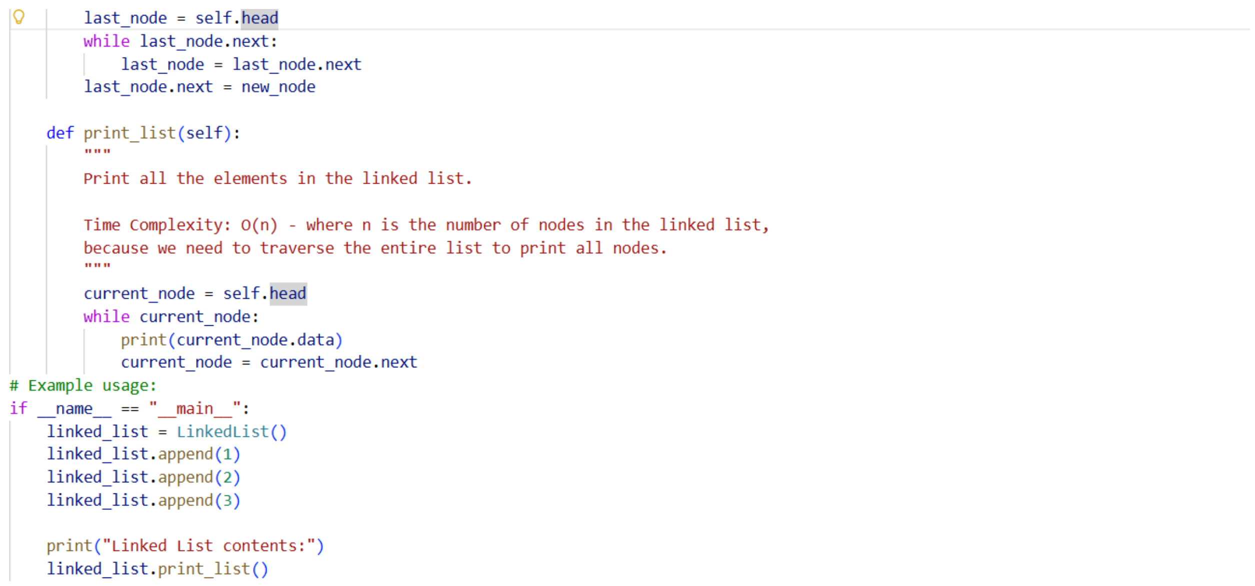
Prompt:

Improve the linked list implementation by adding method-level docstrings and explaining time complexity.

Python Code:

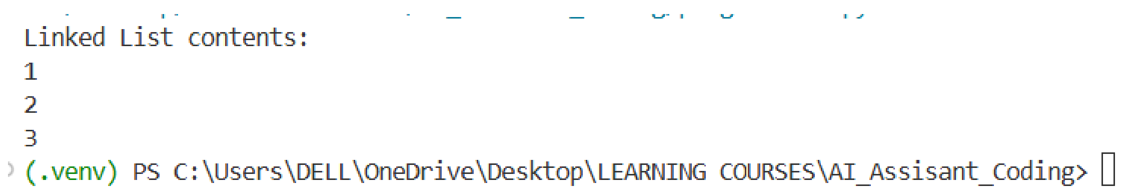


```python
#Improve the linked list implementation by adding method-level docstrings and explaining time complexity.
class Node:
    pass
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        """
        Append a new node with the given data to the end of the linked list.

        Time Complexity: O(n) - where n is the number of nodes in the linked list,
        because we need to traverse the entire list to find the last node.

        Args:
            data: The data to be stored in the new node.
        """
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def print_list(self):
        """
        Print all the elements in the linked list.

        Time Complexity: O(n) - where n is the number of nodes in the linked list,
        because we need to traverse the entire list to print all nodes.
        """
```



```python
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def print_list(self):
        """
        Print all the elements in the linked list.

        Time Complexity: O(n) - where n is the number of nodes in the linked list,
        because we need to traverse the entire list to print all nodes.
        """
        current_node = self.head
        while current_node:
            print(current_node.data)
            current_node = current_node.next
# Example usage:
if __name__ == "__main__":
    linked_list = LinkedList()
    linked_list.append(1)
    linked_list.append(2)
    linked_list.append(3)

    print("Linked List contents:")
    linked_list.print_list()
```

Output:


```
Linked List contents:
1
2
3
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
```

Justification:

1.A singly linked list consists of nodes connected using pointers.

2.Each node contains data and a reference to the next node.

3.insert() adds a new node (typically at the end).

4.display() traverses the list from head to end.

5.Dynamic memory allocation allows flexible size management.

Task Description 4 : Binary Search Tree (BST)

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

Sample Input Code:

class BST:

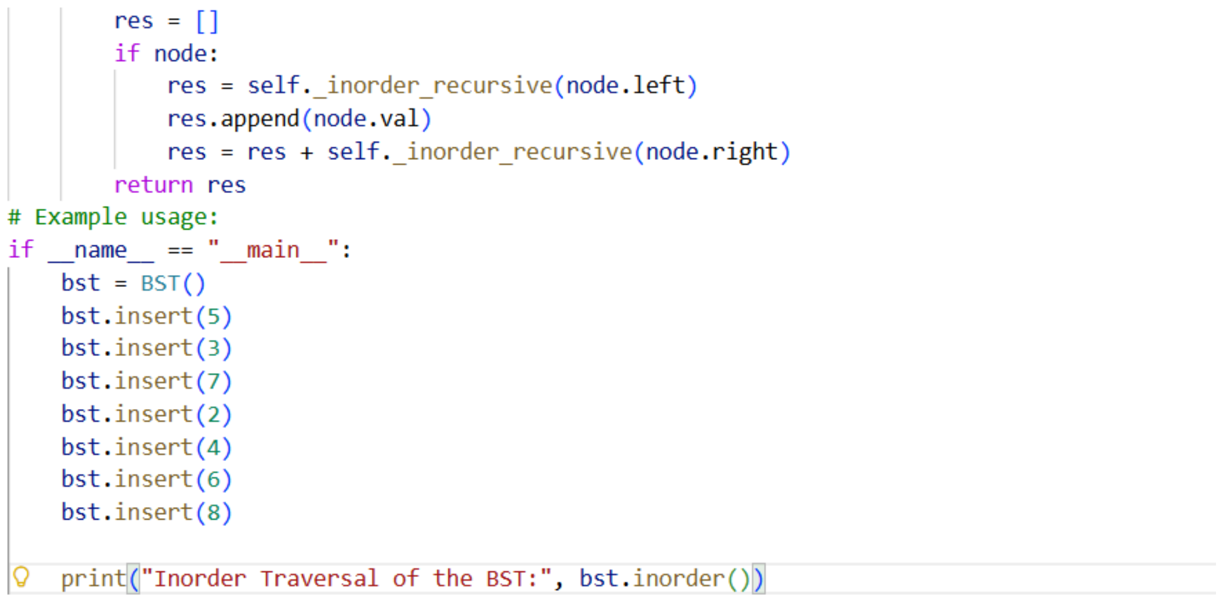
pass

Prompt:

Create a Binary Search Tree class in Python with recursive `insert()` and `inorder()` traversal methods.
Ensure proper object-oriented structure and add docstrings.

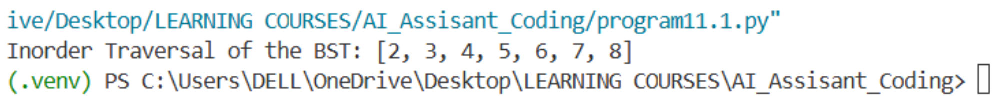
Python Code :

```python
#Create a Binary Search Tree class in Python with recursive insert() and inorder() traversal
class BST:
    pass
    class Node:
        def __init__(self, key):
            self.left = None
            self.right = None
            self.val = key
    def __init__(self):
        self.root = None
    def insert(self, key):
        """Inserts a new key into the BST."""
        if self.root is None:
            self.root = self.Node(key)
        else:
            self._insert_recursive(self.root, key)
    def _insert_recursive(self, node, key):
        """Helper method to insert a new key recursively."""
        if key < node.val:
            if node.left is None:
                node.left = self.Node(key)
            else:
                self._insert_recursive(node.left, key)
        else:
            if node.right is None:
                node.right = self.Node(key)
            else:
                self._insert_recursive(node.right, key)
    def inorder(self):
        """Performs an inorder traversal of the BST and returns a list of values."""
        return self._inorder_recursive(self.root)
    def _inorder_recursive(self, node):
        """Helper method to perform an inorder traversal recursively."""
        res = []
        if node:
            res = self._inorder_recursive(node.left)
```



```
        res = []
        if node:
            res = self._inorder_recursive(node.left)
            res.append(node.val)
            res = res + self._inorder_recursive(node.right)
        return res
# Example usage:
if __name__ == "__main__":
    bst = BST()
    bst.insert(5)
    bst.insert(3)
    bst.insert(7)
    bst.insert(2)
    bst.insert(4)
    bst.insert(6)
    bst.insert(8)

    print("Inorder Traversal of the BST:", bst.inorder())
```

Output:



```
ive/Desktop/LEARNING COURSES/AI_Assisant_Coding/program11.1.py"
Inorder Traversal of the BST: [2, 3, 4, 5, 6, 7, 8]
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
```

Justification:

1.BST is a hierarchical data structure with left and right children.

2.Left subtree contains values smaller than the root.

3.Right subtree contains values greater than the root.

4.Recursive insert() maintains BST property.

5.In-order traversal prints elements in sorted order.

Task Description 5 : Hash Table

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

delete methods.

Sample Input Code:

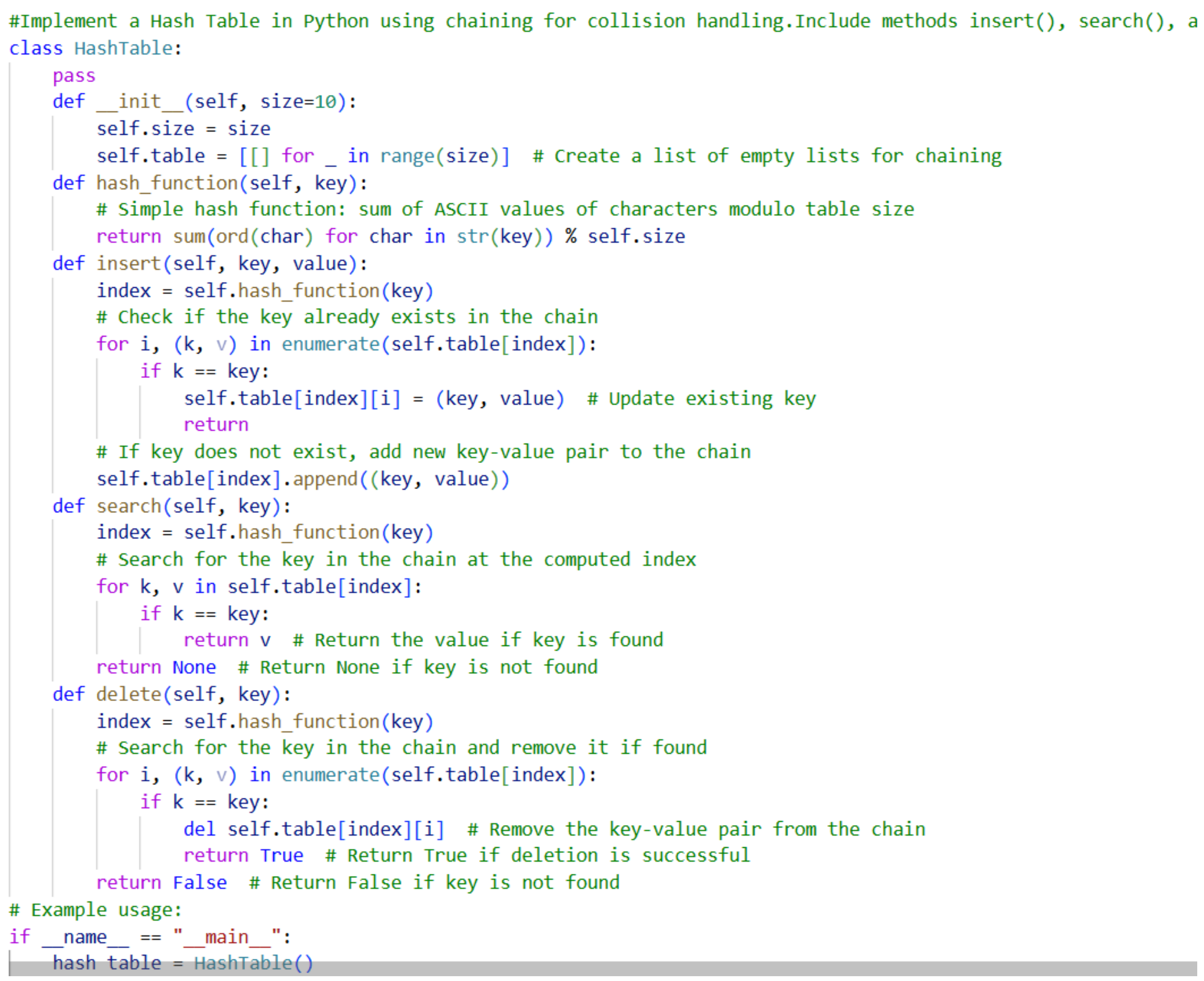
class HashTable:

pass

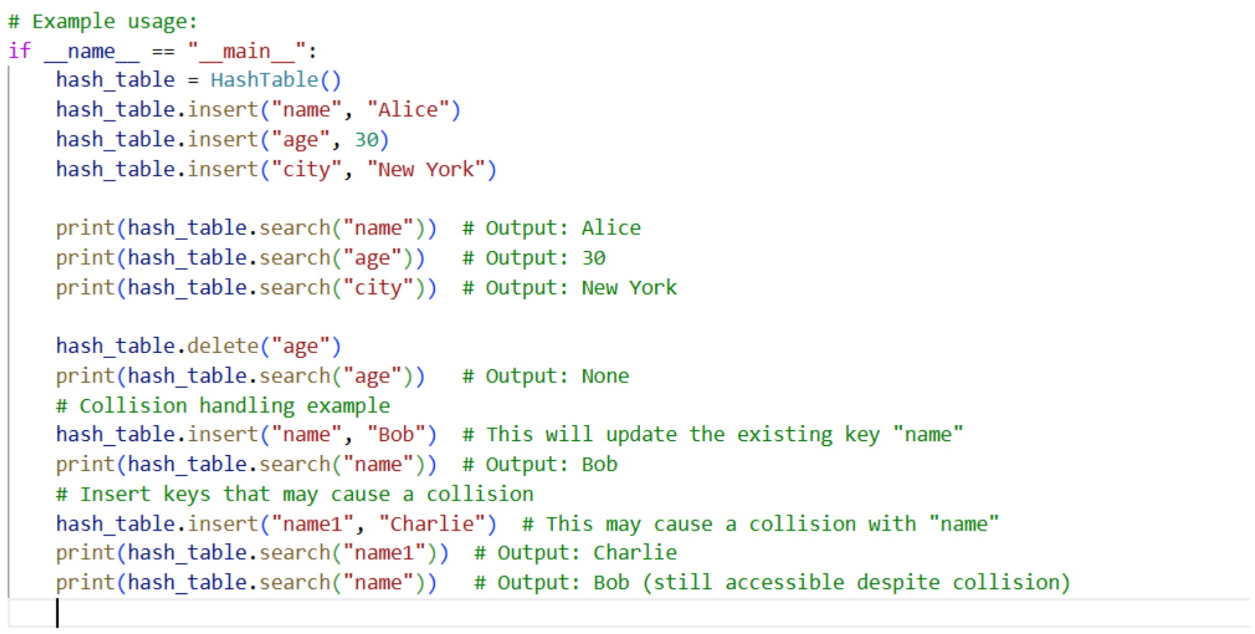
Prompt:

Implement a Hash Table in Python using chaining for collision handling.
Include methods `insert()`, `search()`, and `delete()`.
Use a simple hash function and explain collision handling clearly in comments.

Python Code :

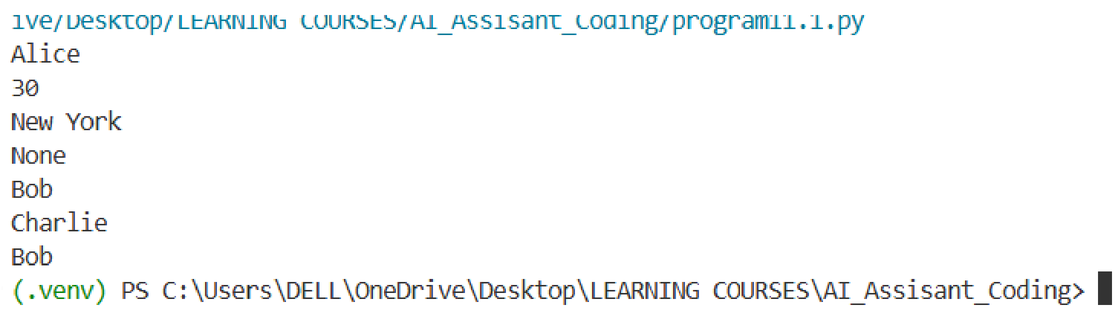


```python
#Implement a Hash Table in Python using chaining for collision handling.Include methods insert(), search(), a
class HashTable:
    pass
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]  # Create a list of empty lists for chaining
    def hash_function(self, key):
        # Simple hash function: sum of ASCII values of characters modulo table size
        return sum(ord(char) for char in str(key)) % self.size
    def insert(self, key, value):
        index = self.hash_function(key)
        # Check if the key already exists in the chain
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        # If key does not exist, add new key-value pair to the chain
        self.table[index].append((key, value))
    def search(self, key):
        index = self.hash_function(key)
        # Search for the key in the chain at the computed index
        for k, v in self.table[index]:
            if k == key:
                return v  # Return the value if key is found
        return None  # Return None if key is not found
    def delete(self, key):
        index = self.hash_function(key)
        # Search for the key in the chain and remove it if found
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]  # Remove the key-value pair from the chain
                return True  # Return True if deletion is successful
        return False  # Return False if key is not found
# Example usage:
if __name__ == "__main__":
    hash_table = HashTable()
```



```python
# Example usage:
if __name__ == "__main__":
    hash_table = HashTable()
    hash_table.insert("name", "Alice")
    hash_table.insert("age", 30)
    hash_table.insert("city", "New York")

    print(hash_table.search("name"))  # Output: Alice
    print(hash_table.search("age"))   # Output: 30
    print(hash_table.search("city"))  # Output: New York

    hash_table.delete("age")
    print(hash_table.search("age"))   # Output: None
    # Collision handling example
    hash_table.insert("name", "Bob")  # This will update the existing key "name"
    print(hash_table.search("name"))  # Output: Bob
    # Insert keys that may cause a collision
    hash_table.insert("name1", "Charlie")  # This may cause a collision with "name"
    print(hash_table.search("name1"))  # Output: Charlie
    print(hash_table.search("name"))   # Output: Bob (still accessible despite collision)
```

Output:



```
ive/Desktop/LEARNING COURSES/AI_Assisant_Coding/program11.1.py
Alice
30
New York
None
Bob
Charlie
Bob
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
```

Justification:

1.Hash table stores data using key-value pairs.

2.A hash function converts keys into index positions.

3.Chaining handles collisions using lists at each index.

4.search() retrieves values quickly using keys.

5.delete() removes key-value pairs efficiently.

Task Description 6 : Graph Representation

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

Sample Input Code:

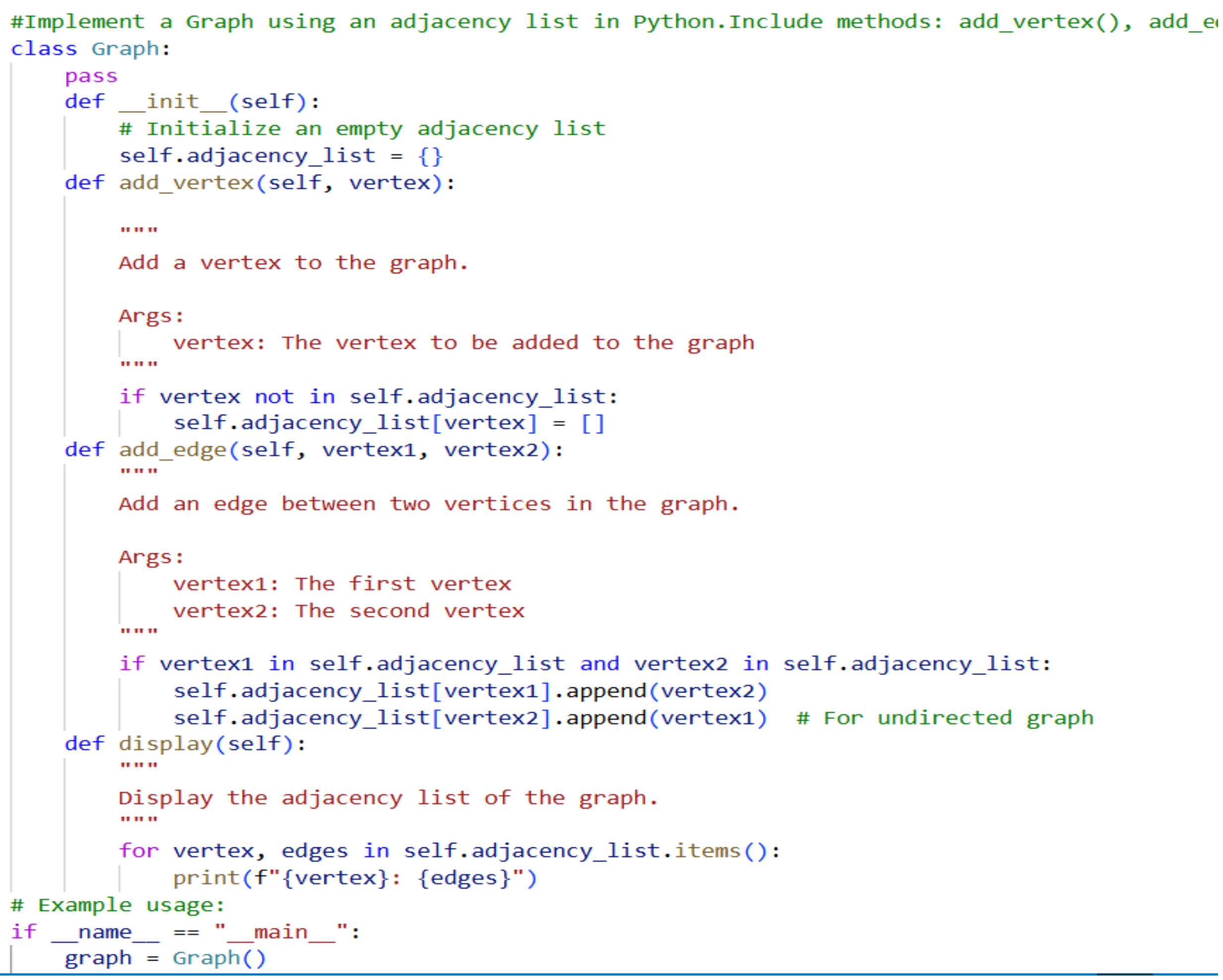
class Graph:

pass

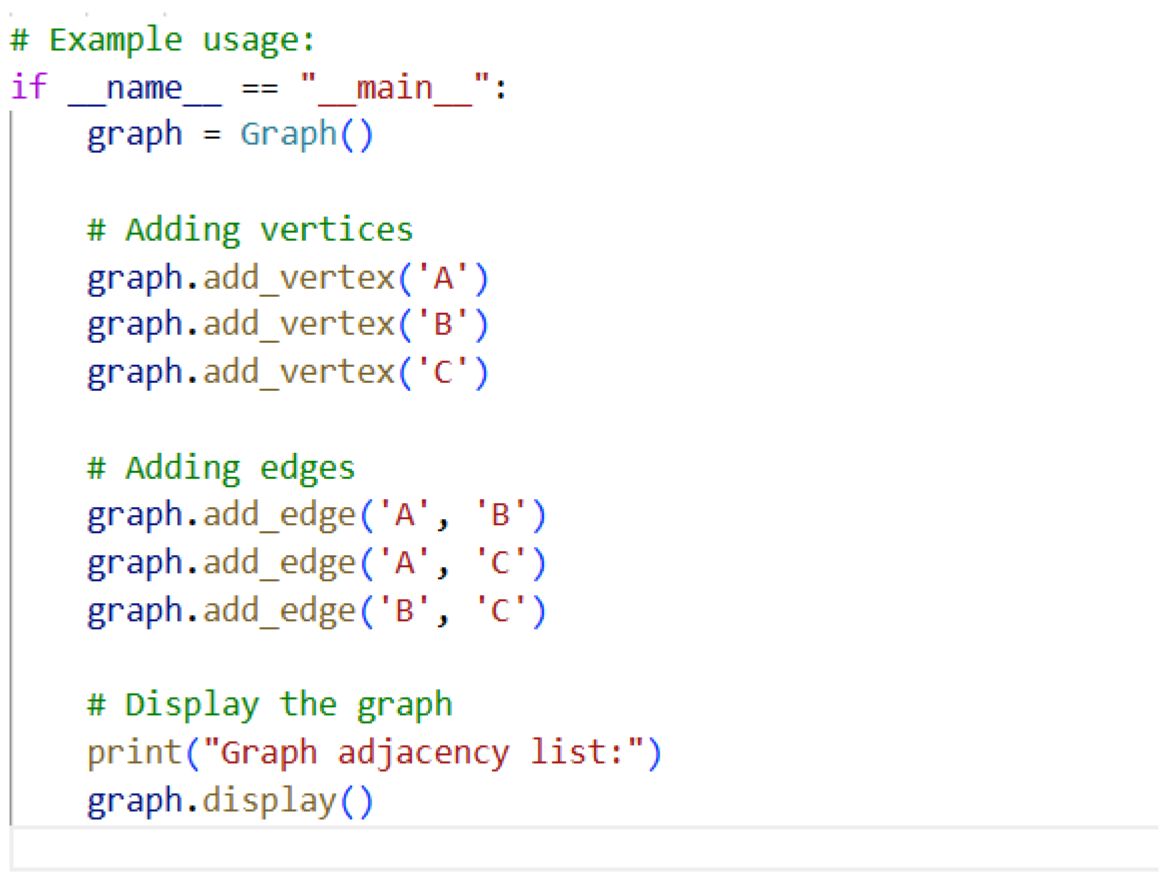
Prompt:

Implement a Graph using an adjacency list in Python.
Include methods: `add_vertex()`, `add_edge()`, and `display()`.
Add clear comments explaining how adjacency lists work.

Python Code :

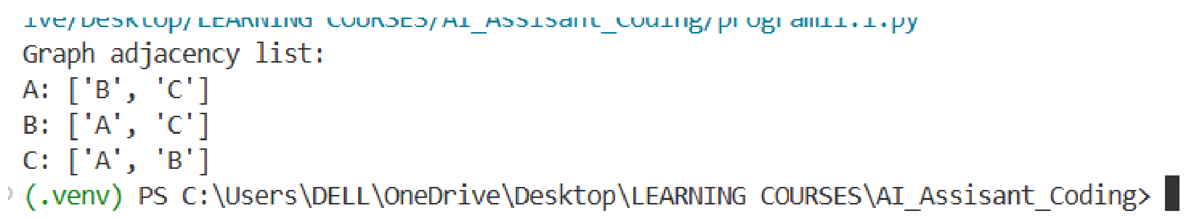


```python
#Implement a Graph using an adjacency list in Python.Include methods: add_vertex(), add_e
class Graph:
    pass
    def __init__(self):
        # Initialize an empty adjacency list
        self.adjacency_list = {}
    def add_vertex(self, vertex):

        """
        Add a vertex to the graph.

        Args:
            vertex: The vertex to be added to the graph
        """
        if vertex not in self.adjacency_list:
            self.adjacency_list[vertex] = []
    def add_edge(self, vertex1, vertex2):
        """
        Add an edge between two vertices in the graph.

        Args:
            vertex1: The first vertex
            vertex2: The second vertex
        """
        if vertex1 in self.adjacency_list and vertex2 in self.adjacency_list:
            self.adjacency_list[vertex1].append(vertex2)
            self.adjacency_list[vertex2].append(vertex1)  # For undirected graph
    def display(self):
        """
        Display the adjacency list of the graph.
        """
        for vertex, edges in self.adjacency_list.items():
            print(f"{vertex}: {edges}")
# Example usage:
if __name__ == "__main__":
    graph = Graph()
```



```python
# Example usage:
if __name__ == "__main__":
    graph = Graph()

    # Adding vertices
    graph.add_vertex('A')
    graph.add_vertex('B')
    graph.add_vertex('C')

    # Adding edges
    graph.add_edge('A', 'B')
    graph.add_edge('A', 'C')
    graph.add_edge('B', 'C')

    # Display the graph
    print("Graph adjacency list:")
    graph.display()
```

Output:



```
ive/Desktop/LEARNING COURSES/AI_Assisant_Coding/program11.1.py
Graph adjacency list:
A: ['B', 'C']
B: ['A', 'C']
C: ['A', 'B']
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
```

Justification:

1.Graph consists of vertices (nodes) and edges (connections).

2.Adjacency list stores neighbors of each vertex.

3.add_vertex() creates a new node in the graph.

4.add_edge() connects two vertices.

5.Efficient for representing sparse graphs.

Task Description 7 : Priority Queue

Task: Use AI to implement a priority queue using Python’s heapq

module

Sample Input Code:

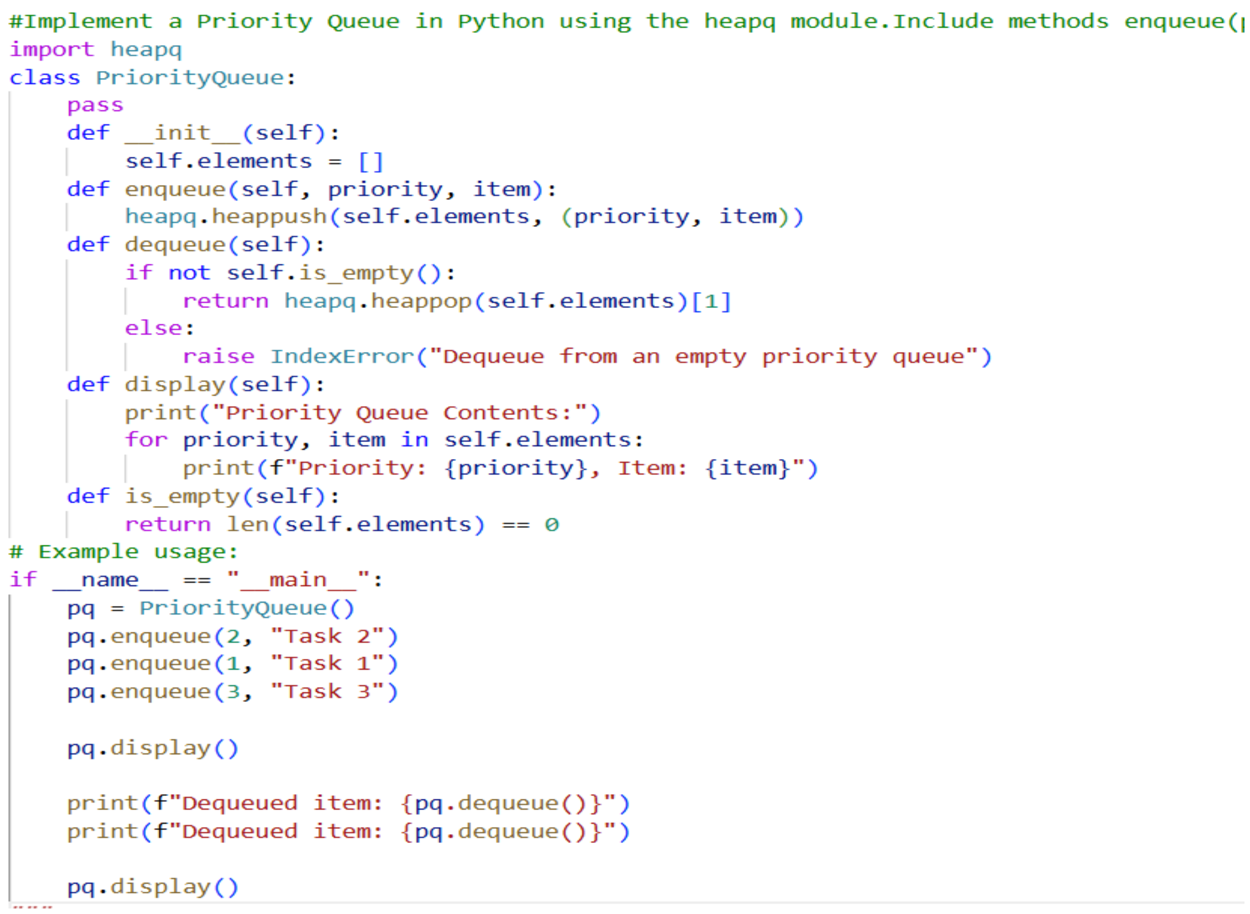
class PriorityQueue:

pass

Prompt:

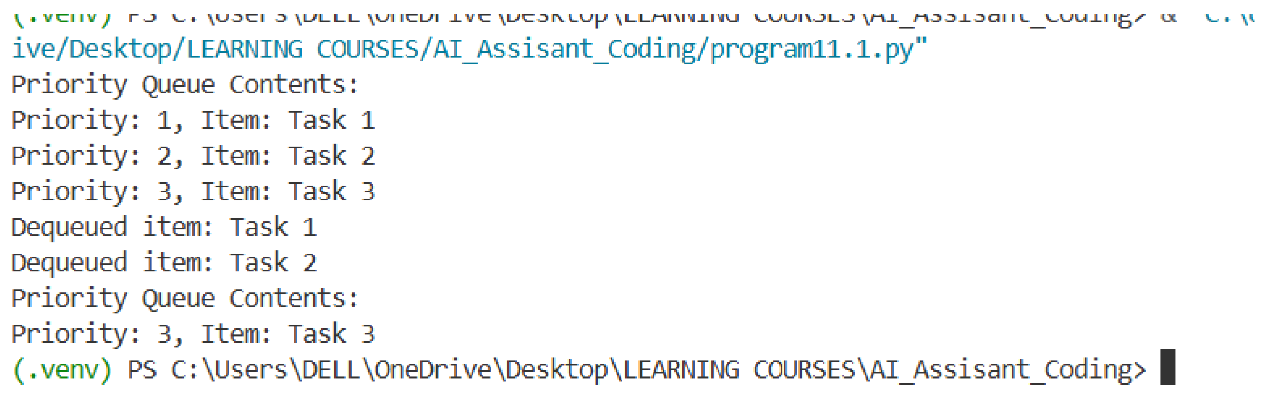
Implement a Priority Queue in Python using the `heapq` module.
Include methods `enqueue(priority, item)`, `dequeue()`, and `display()`.
Add documentation explaining how a min-heap works.

Python Code :



```python
#Implement a Priority Queue in Python using the heapq module.Include methods enqueue(
import heapq
class PriorityQueue:
    pass
    def __init__(self):
        self.elements = []
    def enqueue(self, priority, item):
        heapq.heappush(self.elements, (priority, item))
    def dequeue(self):
        if not self.is_empty():
            return heapq.heappop(self.elements)[1]
        else:
            raise IndexError("Dequeue from an empty priority queue")
    def display(self):
        print("Priority Queue Contents:")
        for priority, item in self.elements:
            print(f"Priority: {priority}, Item: {item}")
    def is_empty(self):
        return len(self.elements) == 0
# Example usage:
if __name__ == "__main__":
    pq = PriorityQueue()
    pq.enqueue(2, "Task 2")
    pq.enqueue(1, "Task 1")
    pq.enqueue(3, "Task 3")

    pq.display()

    print(f"Dequeued item: {pq.dequeue()}")
    print(f"Dequeued item: {pq.dequeue()}")

    pq.display()
```

Output:



```
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding> & "c:/
ive/Desktop/LEARNING COURSES/AI_Assisant_Coding/program11.1.py"
Priority Queue Contents:
Priority: 1, Item: Task 1
Priority: 2, Item: Task 2
Priority: 3, Item: Task 3
Dequeued item: Task 1
Dequeued item: Task 2
Priority Queue Contents:
Priority: 3, Item: Task 3
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
```

Justification:

1.Priority Queue removes elements based on priority, not order.

2.Lower (or higher) priority value determines removal order.

3.enqueue() inserts element with priority.

4.dequeue() removes highest-priority element.

5.Heap structure ensures efficient insertion and deletion.

Task Description 8 : Deque

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

collections.deque.

Sample Input Code:

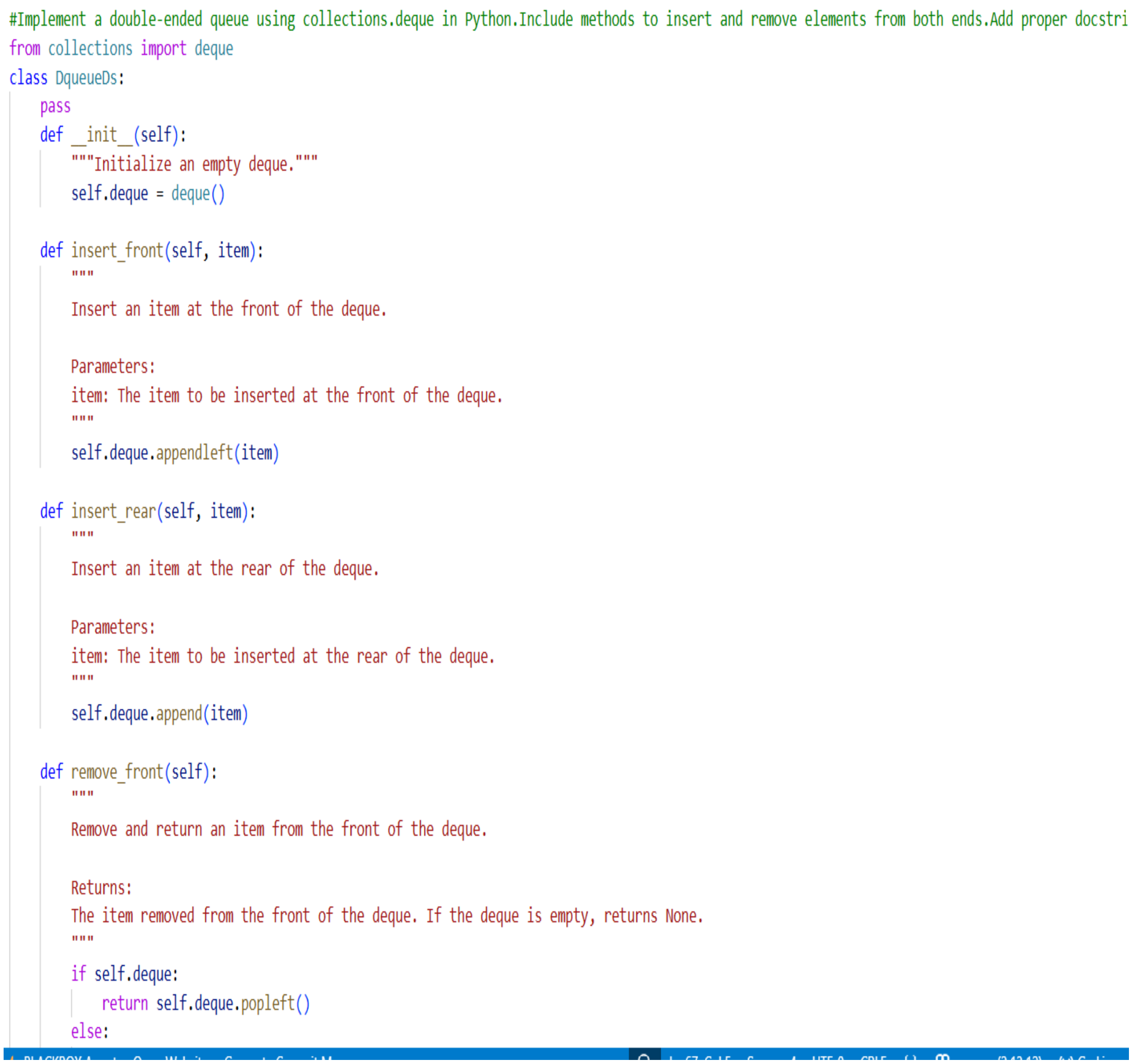
class DequeDS:

pass

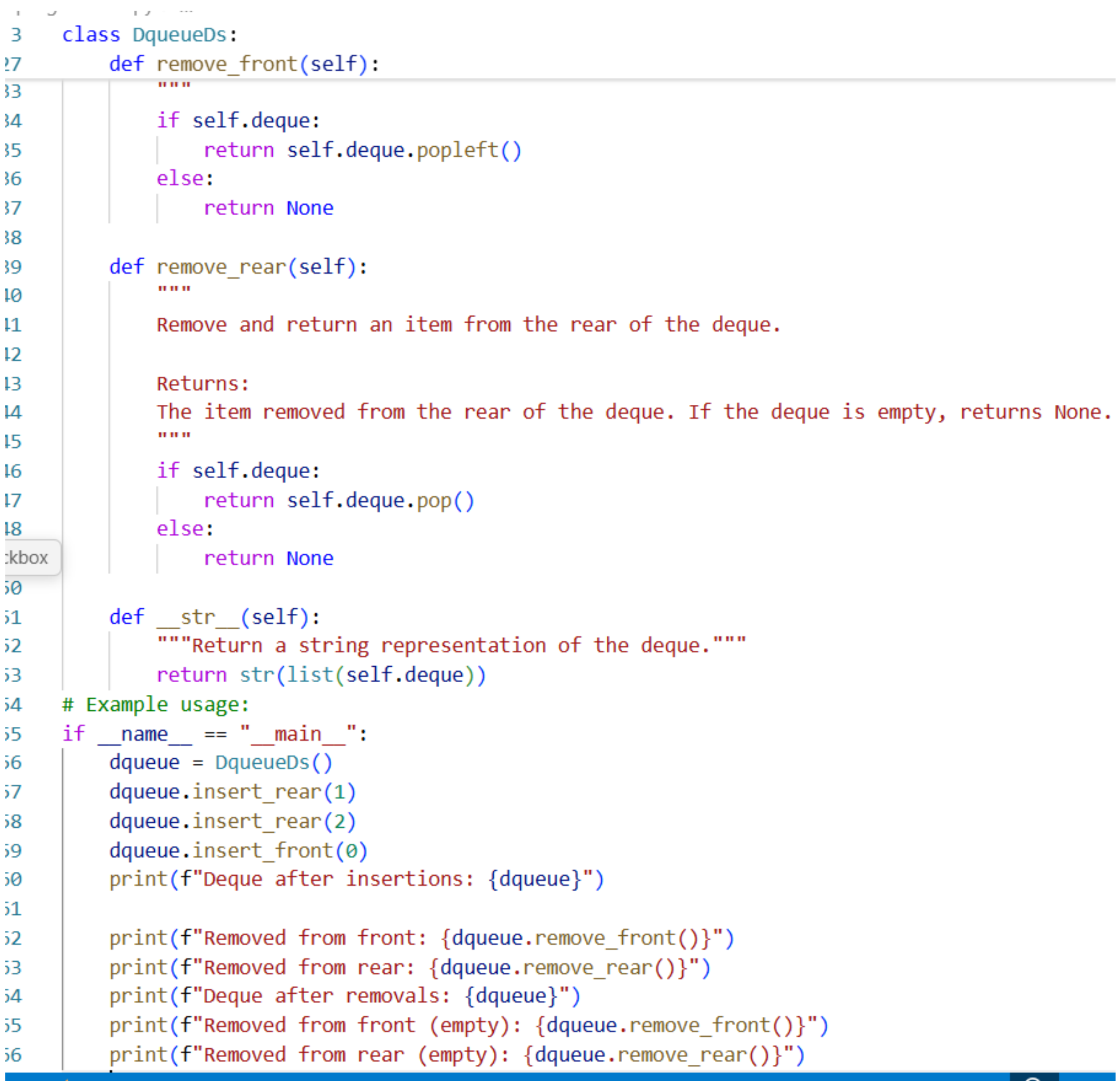
Prompt:

Implement a double-ended queue using `collections.deque` in Python.
Include methods to insert and remove elements from both ends.
Add proper docstrings.

Python Code :

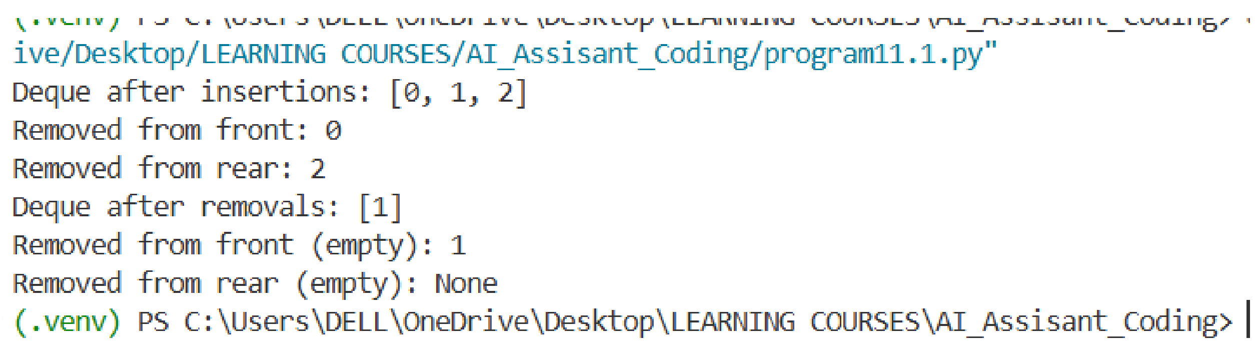


```python
#Implement a double-ended queue using collections.deque in Python.Include methods to insert and remove elements from both ends.Add proper docstri
from collections import deque
class DqueueDs:
    pass
    def __init__(self):
        """Initialize an empty deque."""
        self.deque = deque()

    def insert_front(self, item):
        """
        Insert an item at the front of the deque.

        Parameters:
        item: The item to be inserted at the front of the deque.
        """
        self.deque.appendleft(item)

    def insert_rear(self, item):
        """
        Insert an item at the rear of the deque.

        Parameters:
        item: The item to be inserted at the rear of the deque.
        """
        self.deque.append(item)

    def remove_front(self):
        """
        Remove and return an item from the front of the deque.

        Returns:
        The item removed from the front of the deque. If the deque is empty, returns None.
        """
        if self.deque:
            return self.deque.popleft()
        else:
```



```python
class DqueueDs:
    def remove_front(self):
        """
        if self.deque:
            return self.deque.popleft()
        else:
            return None

    def remove_rear(self):
        """
        Remove and return an item from the rear of the deque.

        Returns:
        The item removed from the rear of the deque. If the deque is empty, returns None.
        """
        if self.deque:
            return self.deque.pop()
        else:
            return None

    def __str__(self):
        """Return a string representation of the deque."""
        return str(list(self.deque))
# Example usage:
if __name__ == "__main__":
    dqueue = DqueueDs()
    dqueue.insert_rear(1)
    dqueue.insert_rear(2)
    dqueue.insert_front(0)
    print(f"Deque after insertions: {dqueue}")

    print(f"Removed from front: {dqueue.remove_front()}")
    print(f"Removed from rear: {dqueue.remove_rear()}")
    print(f"Deque after removals: {dqueue}")
    print(f"Removed from front (empty): {dqueue.remove_front()}")
    print(f"Removed from rear (empty): {dqueue.remove_rear()}")
```

Output:



```
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
ive/Desktop/LEARNING COURSES/AI_Assisant_Coding/program11.1.py"
Deque after insertions: [0, 1, 2]
Removed from front: 0
Removed from rear: 2
Deque after removals: [1]
Removed from front (empty): 1
Removed from rear (empty): None
(.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
```

Justification:

1.Deque allows insertion and deletion at both front and rear.

2.Combines features of stack and queue.

3.Efficient O(1) operations at both ends.

4.Useful for sliding window problems.

5.Flexible data structure for bidirectional processing.

Task Description 9: 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.

Prompt:

Generate a complete Python program for a Campus Resource Management System feature using an appropriate data structure.

Choose ONE of the following features and implement it fully:

1. Student Attendance Tracking

2. Event Registration System

3. Library Book Borrowing

4. Bus Scheduling System

5. Cafeteria Order Queue

Requirements:

- Select the most suitable data structure (Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, or Deque).

- Implement a class-based solution.

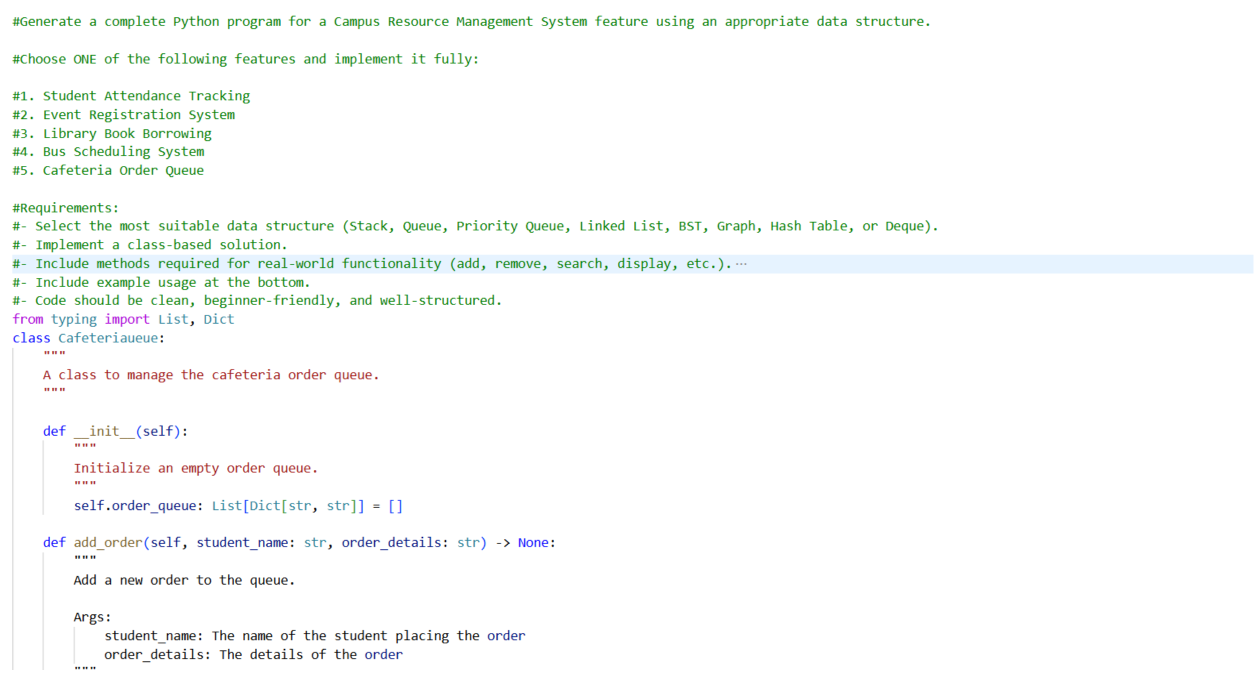
- Include methods required for real-world functionality (add, remove, search, display, etc.).

- Add proper docstrings and comments.

- Include example usage at the bottom.

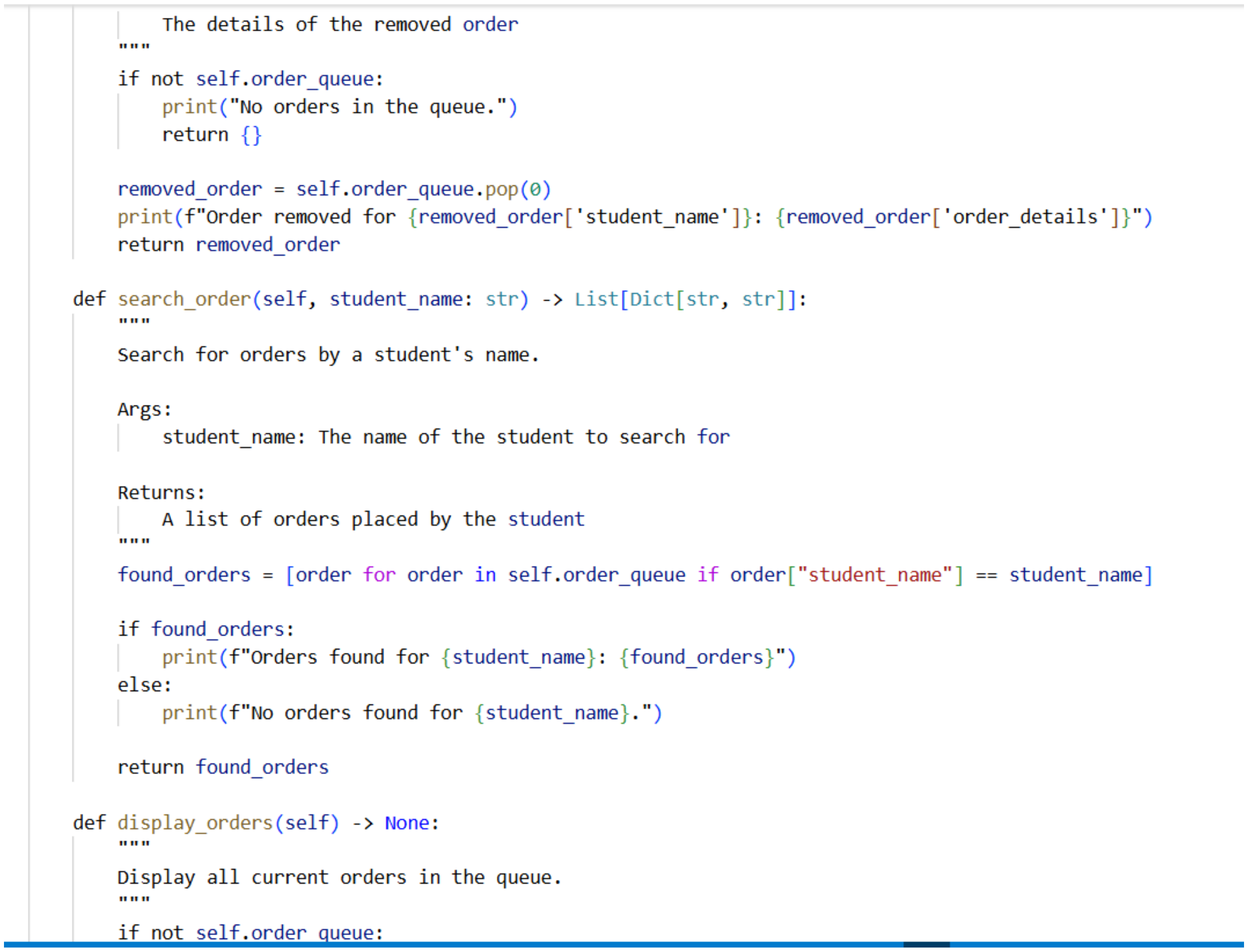
- Code should be clean, beginner-friendly, and well-structured.

Python Code :

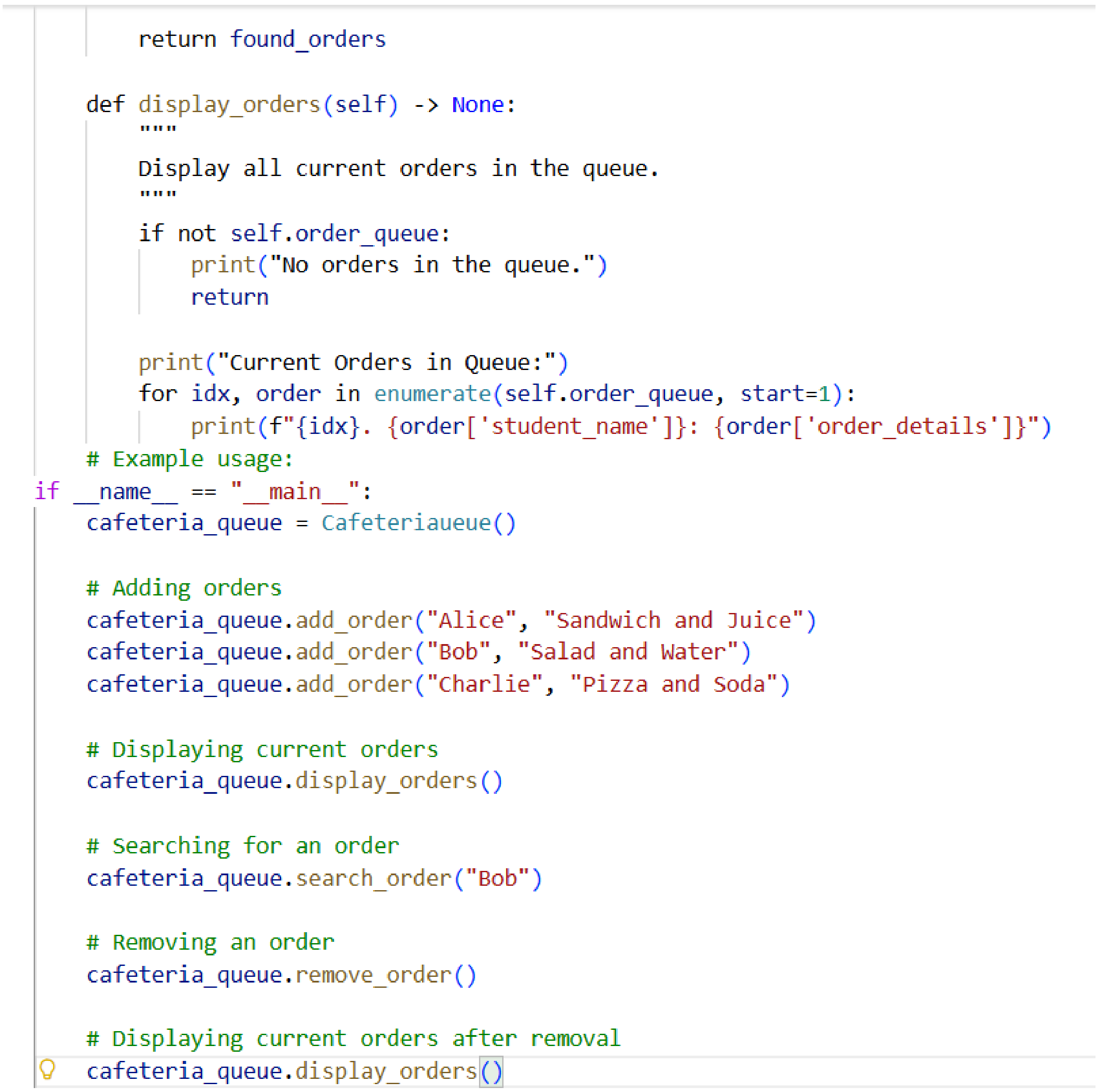


```python
#Generate a complete Python program for a Campus Resource Management System feature using an appropriate data structure.

#Choose ONE of the following features and implement it fully:

#1. Student Attendance Tracking
#2. Event Registration System
#3. Library Book Borrowing
#4. Bus Scheduling System
#5. Cafeteria Order Queue

#Requirements:
#- Select the most suitable data structure (Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, or Deque).
#- Implement a class-based solution.
#- Include methods required for real-world functionality (add, remove, search, display, etc.).…
#- Include example usage at the bottom.
#- Code should be clean, beginner-friendly, and well-structured.
from typing import List, Dict
class Cafeteriaueue:
    """
    A class to manage the cafeteria order queue.
    """

    def __init__(self):
        """
        Initialize an empty order queue.
        """
        self.order_queue: List[Dict[str, str]] = []

    def add_order(self, student_name: str, order_details: str) -> None:
        """
        Add a new order to the queue.

        Args:
            student_name: The name of the student placing the order
            order_details: The details of the order
        """
```

```
def add_order(self, student_name: str, order_details: str) -> None:

    Add a new order to the queue.

    Args:
        student_name: The name of the student placing the order
        order_details: The details of the order
    """
    order = {"student_name": student_name, "order_details": order_details}
    self.order_queue.append(order)
    print(f"Order added for {student_name}: {order_details}")

def remove_order(self) -> Dict[str, str]:
    """
    Remove the next order from the queue (FIFO).

    Returns:
        The details of the removed order
    """
    if not self.order_queue:
        print("No orders in the queue.")
        return {}

    removed_order = self.order_queue.pop(0)
    print(f"Order removed for {removed_order['student_name']}: {removed_order['order_details']}")
    return removed_order

def search_order(self, student_name: str) -> List[Dict[str, str]]:
    """
    Search for orders by a student's name.

    Args:
        student_name: The name of the student to search for

    Returns:
```

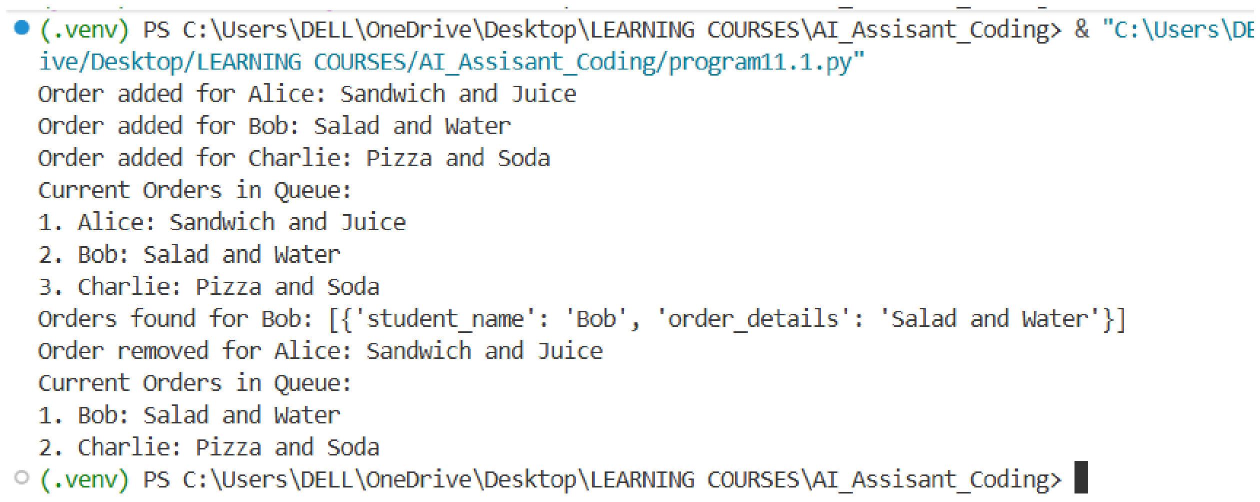


```python
            The details of the removed order
        """
        if not self.order_queue:
            print("No orders in the queue.")
            return {}

        removed_order = self.order_queue.pop(0)
        print(f"Order removed for {removed_order['student_name']}: {removed_order['order_details']}")
        return removed_order

    def search_order(self, student_name: str) -> List[Dict[str, str]]:
        """
        Search for orders by a student's name.

        Args:
            student_name: The name of the student to search for

        Returns:
            A list of orders placed by the student
        """
        found_orders = [order for order in self.order_queue if order["student_name"] == student_name]

        if found_orders:
            print(f"Orders found for {student_name}: {found_orders}")
        else:
            print(f"No orders found for {student_name}.")

        return found_orders

    def display_orders(self) -> None:
        """
        Display all current orders in the queue.
        """
        if not self.order_queue:
```



```
        return found_orders

    def display_orders(self) -> None:
        """
        Display all current orders in the queue.
        """
        if not self.order_queue:
            print("No orders in the queue.")
            return

        print("Current Orders in Queue:")
        for idx, order in enumerate(self.order_queue, start=1):
            print(f"{idx}. {order['student_name']}: {order['order_details']}")
    # Example usage:
if __name__ == "__main__":
    cafeteria_queue = Cafeteriaueue()

    # Adding orders
    cafeteria_queue.add_order("Alice", "Sandwich and Juice")
    cafeteria_queue.add_order("Bob", "Salad and Water")
    cafeteria_queue.add_order("Charlie", "Pizza and Soda")

    # Displaying current orders
    cafeteria_queue.display_orders()

    # Searching for an order
    cafeteria_queue.search_order("Bob")

    # Removing an order
    cafeteria_queue.remove_order()

    # Displaying current orders after removal
    cafeteria_queue.display_orders()
```

Output:



```
● (.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding> & "C:\Users\DE
ive/Desktop/LEARNING COURSES/AI_Assisant_Coding/program11.1.py"
Order added for Alice: Sandwich and Juice
Order added for Bob: Salad and Water
Order added for Charlie: Pizza and Soda
Current Orders in Queue:
1. Alice: Sandwich and Juice
2. Bob: Salad and Water
3. Charlie: Pizza and Soda
Orders found for Bob: [{'student_name': 'Bob', 'order_details': 'Salad and Water'}]
Order removed for Alice: Sandwich and Juice
Current Orders in Queue:
1. Bob: Salad and Water
2. Charlie: Pizza and Soda
○ (.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
```

Justification:

Feature	**Chosen Data Structure**	**Justification**
Student Attendance Tracking	Deque	Students may enter and exit from multiple points. A deque allows insertion and removal from both ends efficiently.
Event Registration System	Hash Table	Quick lookup, insertion, and deletion of participants by ID is essential, which hash tables provide in O(1) average time.
Library Book Borrowing	Binary Search Tree (BST)	BST allows ordered storage of books, fast search by book ID, and easy tracking of due dates through in-order traversal.
Bus Scheduling System	Graph	Routes and stops are naturally modeled as vertices and edges, enabling path finding and route management.
Cafeteria Order Queue	Queue	Orders must be served in the order they arrive; a queue ensures FIFO behavior perfectly.

Task Description 10: Smart E-Commerce Platform – Data Structure Challenge

An e-commerce company wants to build a Smart Online Shopping System

with:

1. Shopping Cart Management – Add and remove products

dynamically.

2. Order Processing System – Orders processed in the order they are

placed.

3. Top-Selling Products Tracker – Products ranked by sales count.

4. Product Search Engine – Fast lookup of products using product ID.

5. Delivery Route Planning – Connect warehouses and delivery

locations.

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.

Prompt:

Generate a Python program for one feature of a Smart Online Shopping System.

Features to choose from:

1. Shopping Cart Management

2. Order Processing System

3. Top-Selling Products Tracker

4. Product Search Engine

5. Delivery Route Planning

Requirements:

- Choose the most appropriate data structure (Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, or Deque)

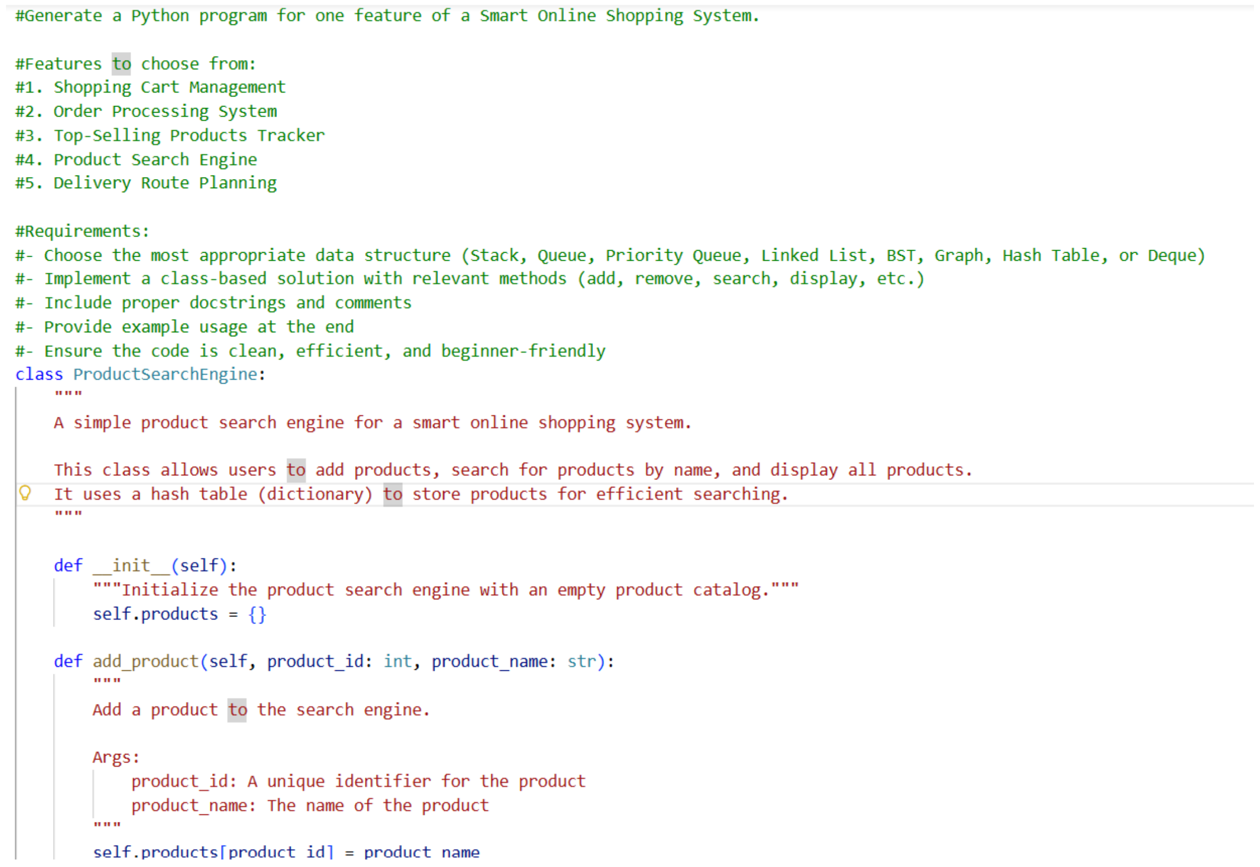
- Implement a class-based solution with relevant methods (add, remove, search, display, etc.)

- Include proper docstrings and comments

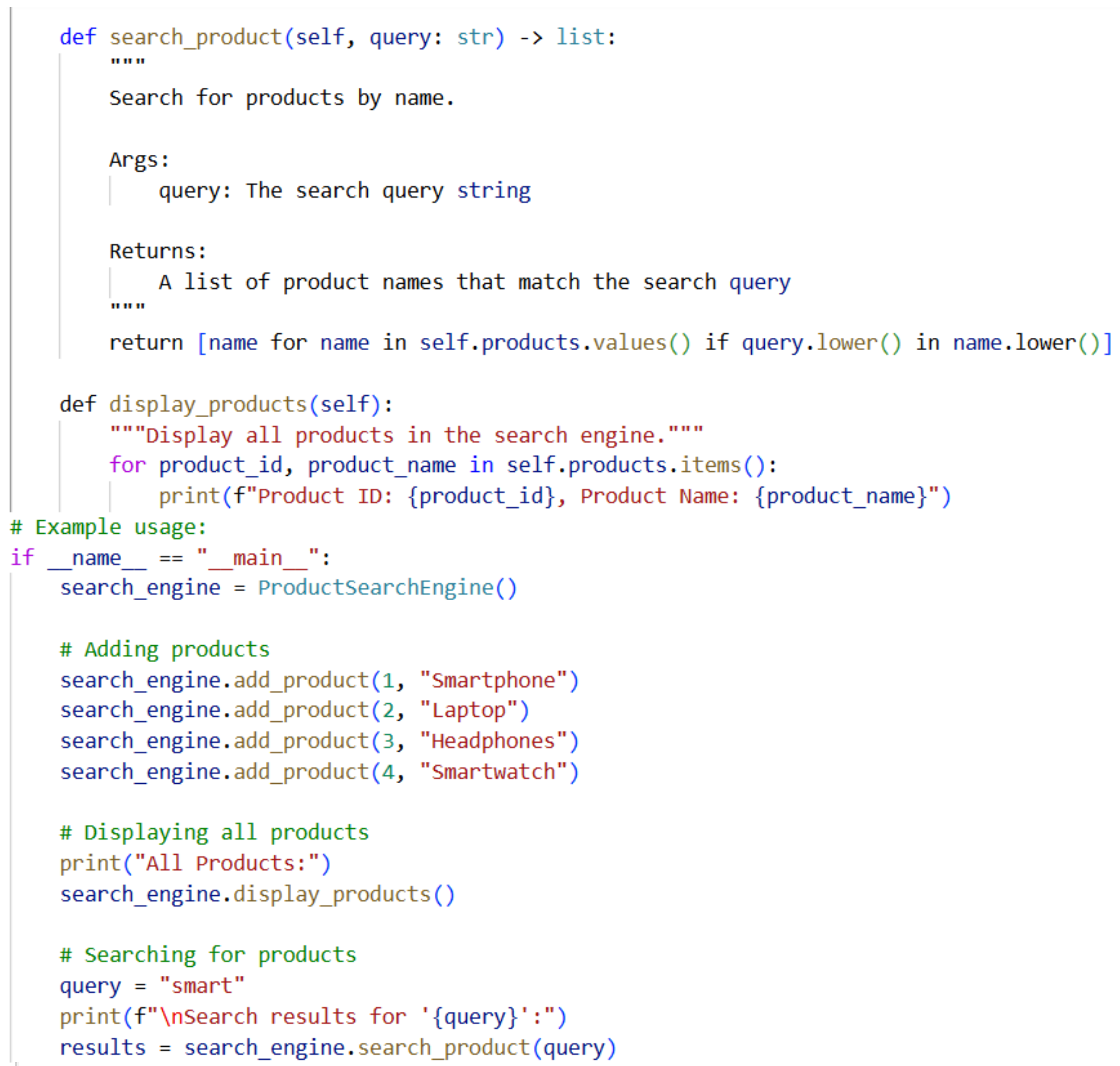
- Provide example usage at the end

- Ensure the code is clean, efficient, and beginner-friendly

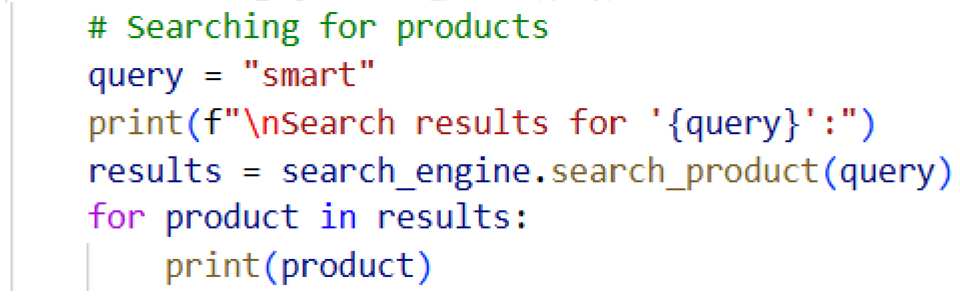
Python Code :



```
#Generate a Python program for one feature of a Smart Online Shopping System.

#Features to choose from:
#1. Shopping Cart Management
#2. Order Processing System
#3. Top-Selling Products Tracker
#4. Product Search Engine
#5. Delivery Route Planning

#Requirements:
#- Choose the most appropriate data structure (Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, or Deque)
#- Implement a class-based solution with relevant methods (add, remove, search, display, etc.)
#- Include proper docstrings and comments
#- Provide example usage at the end
#- Ensure the code is clean, efficient, and beginner-friendly
class ProductSearchEngine:
    """
    A simple product search engine for a smart online shopping system.

    This class allows users to add products, search for products by name, and display all products.
    It uses a hash table (dictionary) to store products for efficient searching.
    """

    def __init__(self):
        """Initialize the product search engine with an empty product catalog."""
        self.products = {}

    def add_product(self, product_id: int, product_name: str):
        """
        Add a product to the search engine.

        Args:
            product_id: A unique identifier for the product
            product_name: The name of the product
        """
        self.products[product_id] = product_name
```

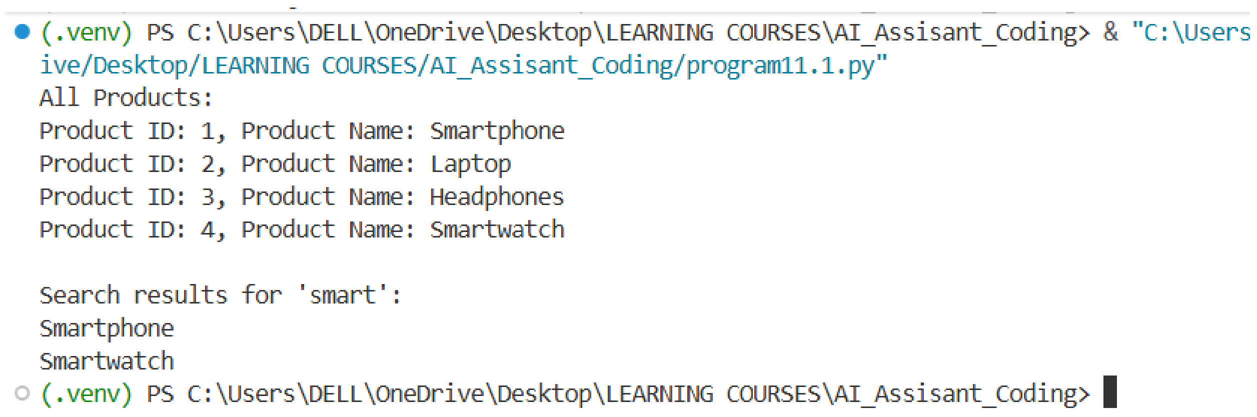


```python
    def search_product(self, query: str) -> list:
        """
        Search for products by name.

        Args:
            query: The search query string

        Returns:
            A list of product names that match the search query
        """
        return [name for name in self.products.values() if query.lower() in name.lower()]

    def display_products(self):
        """Display all products in the search engine."""
        for product_id, product_name in self.products.items():
            print(f"Product ID: {product_id}, Product Name: {product_name}")
# Example usage:
if __name__ == "__main__":
    search_engine = ProductSearchEngine()

    # Adding products
    search_engine.add_product(1, "Smartphone")
    search_engine.add_product(2, "Laptop")
    search_engine.add_product(3, "Headphones")
    search_engine.add_product(4, "Smartwatch")

    # Displaying all products
    print("All Products:")
    search_engine.display_products()

    # Searching for products
    query = "smart"
    print(f"\nSearch results for '{query}':")
    results = search_engine.search_product(query)
```



```python
    # Searching for products
    query = "smart"
    print(f"\nSearch results for '{query}':")
    results = search_engine.search_product(query)
    for product in results:
        print(product)
```

Output:



```
● (.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding> & "C:\Users
  ive/Desktop/LEARNING COURSES/AI_Assisant_Coding/program11.1.py"
  All Products:
  Product ID: 1, Product Name: Smartphone
  Product ID: 2, Product Name: Laptop
  Product ID: 3, Product Name: Headphones
  Product ID: 4, Product Name: Smartwatch

  Search results for 'smart':
  Smartphone
  Smartwatch
○ (.venv) PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding>
```

Justification:

Feature	Chosen Data Structure	Justification
Shopping Cart Management	Linked List	Users can dynamically add/remove items; linked lists allow flexible insertion and deletion without shifting elements.
Order Processing System	Queue	Orders are fulfilled in the sequence they are placed, making a FIFO queue the natural choice.
Top-Selling Products Tracker	Priority Queue	Products need to be ranked by sales count; a priority queue allows efficient retrieval of top-selling items.
Product Search Engine	Hash Table	Fast lookup of products by ID is critical; hash tables provide O(1) average search time.
Delivery Route Planning	Graph	Warehouses and delivery locations are nodes; roads/routes are edges. Graphs allow route optimization and pathfinding.