AI ASSISTED CODING

LAB ASSIGNMENT-11.1

Roll no: 2403A510D1

Name: Likhitha Pothunuri

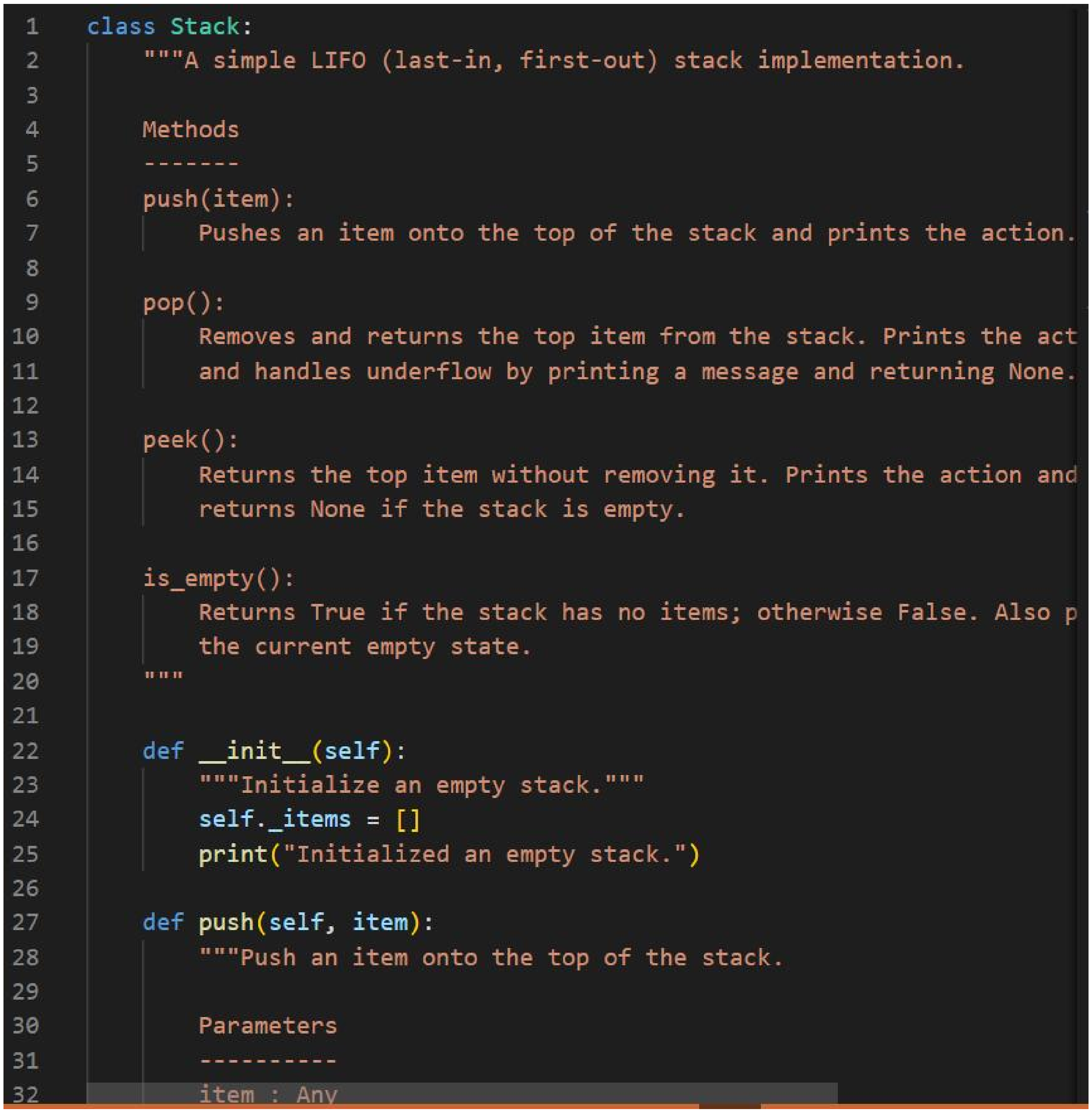
2nd year CSE

Task1:

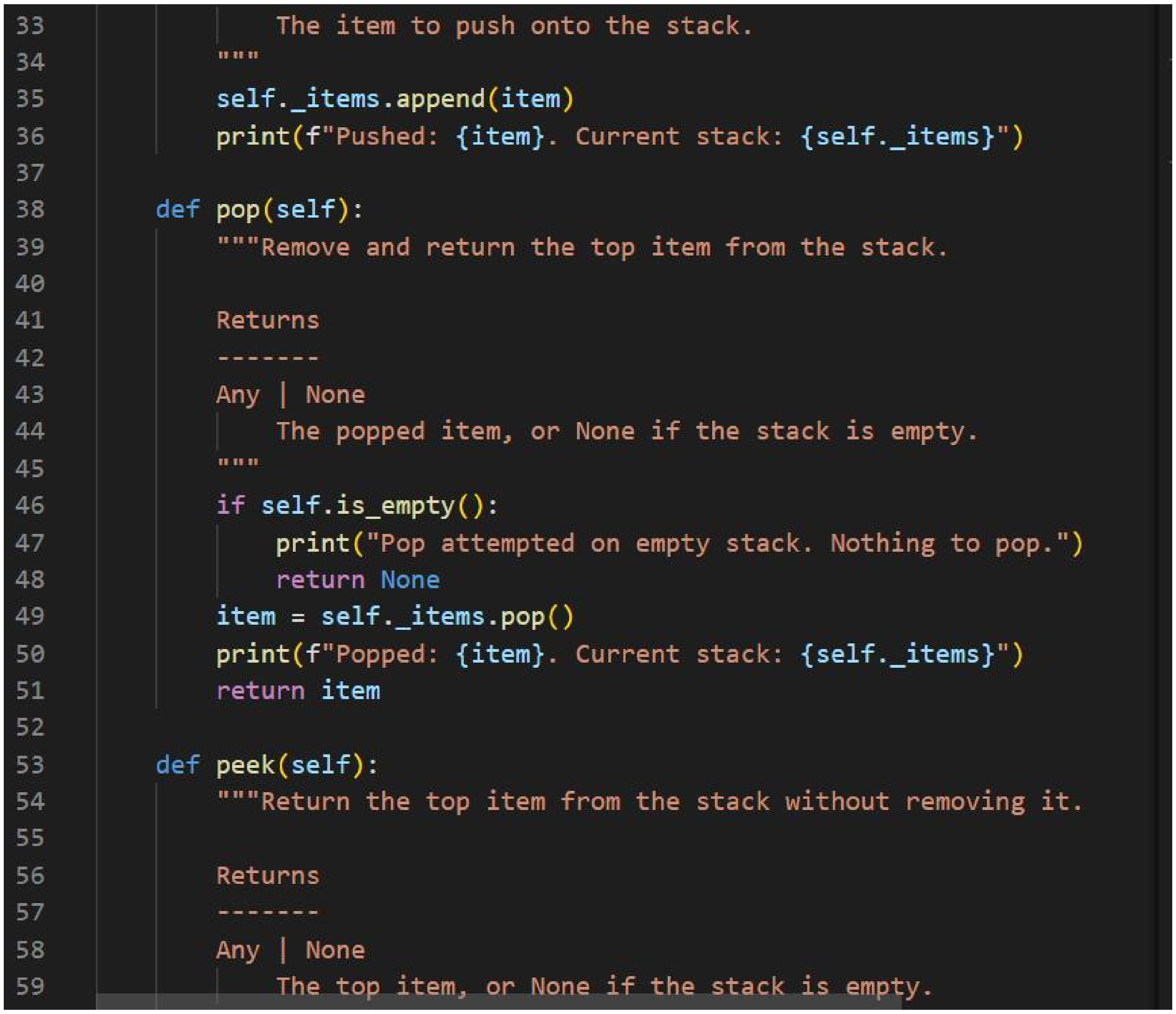
Prompt:

Generate a Stack class with push, pop, peek, and is_empty
methods.Sample input should be like class Stack:pass
Sample output should be like A functional stack implementation with all required methods anddocstrings along with print statements and it should take input from user and then give output as expected .

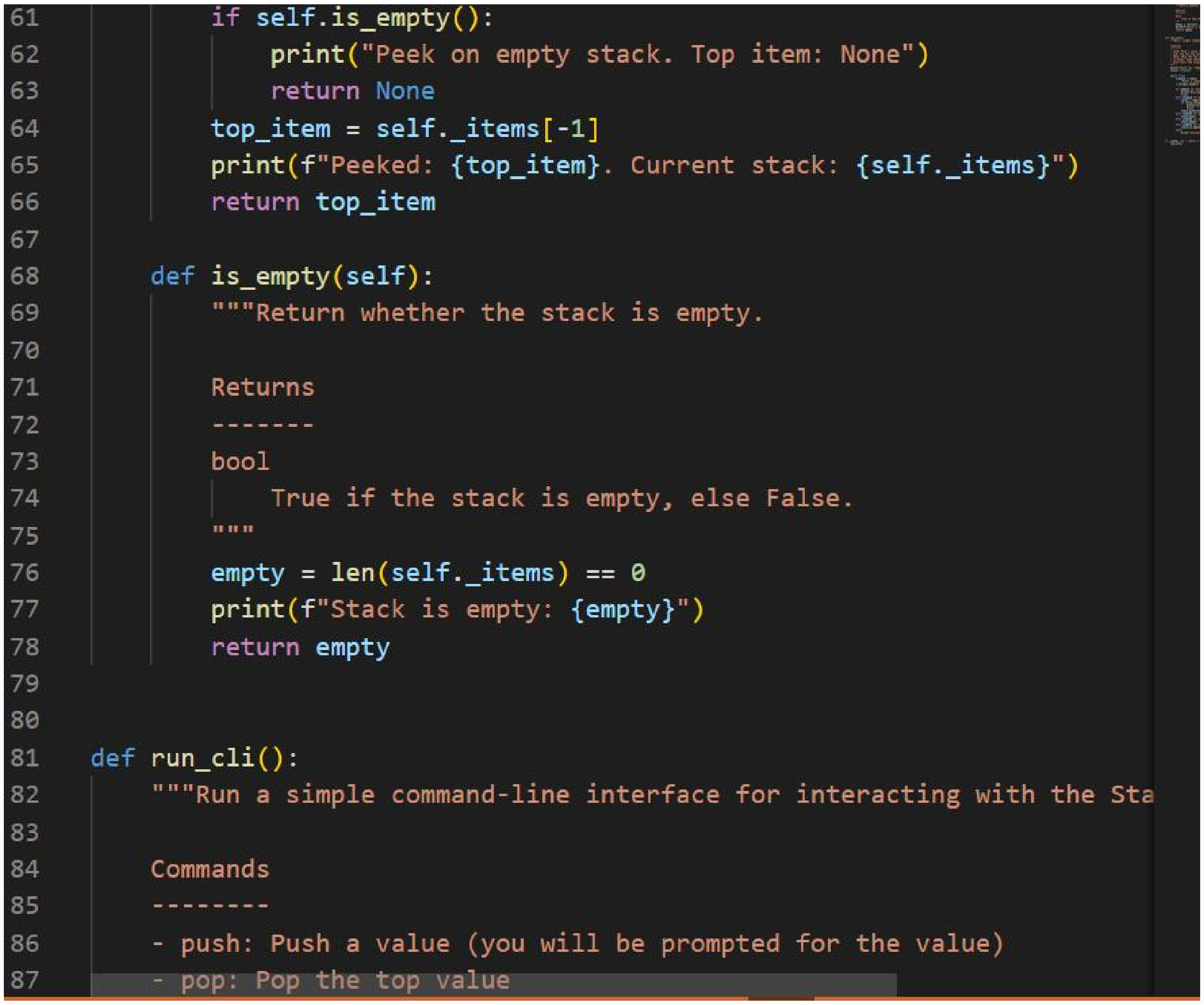
Code Generated:



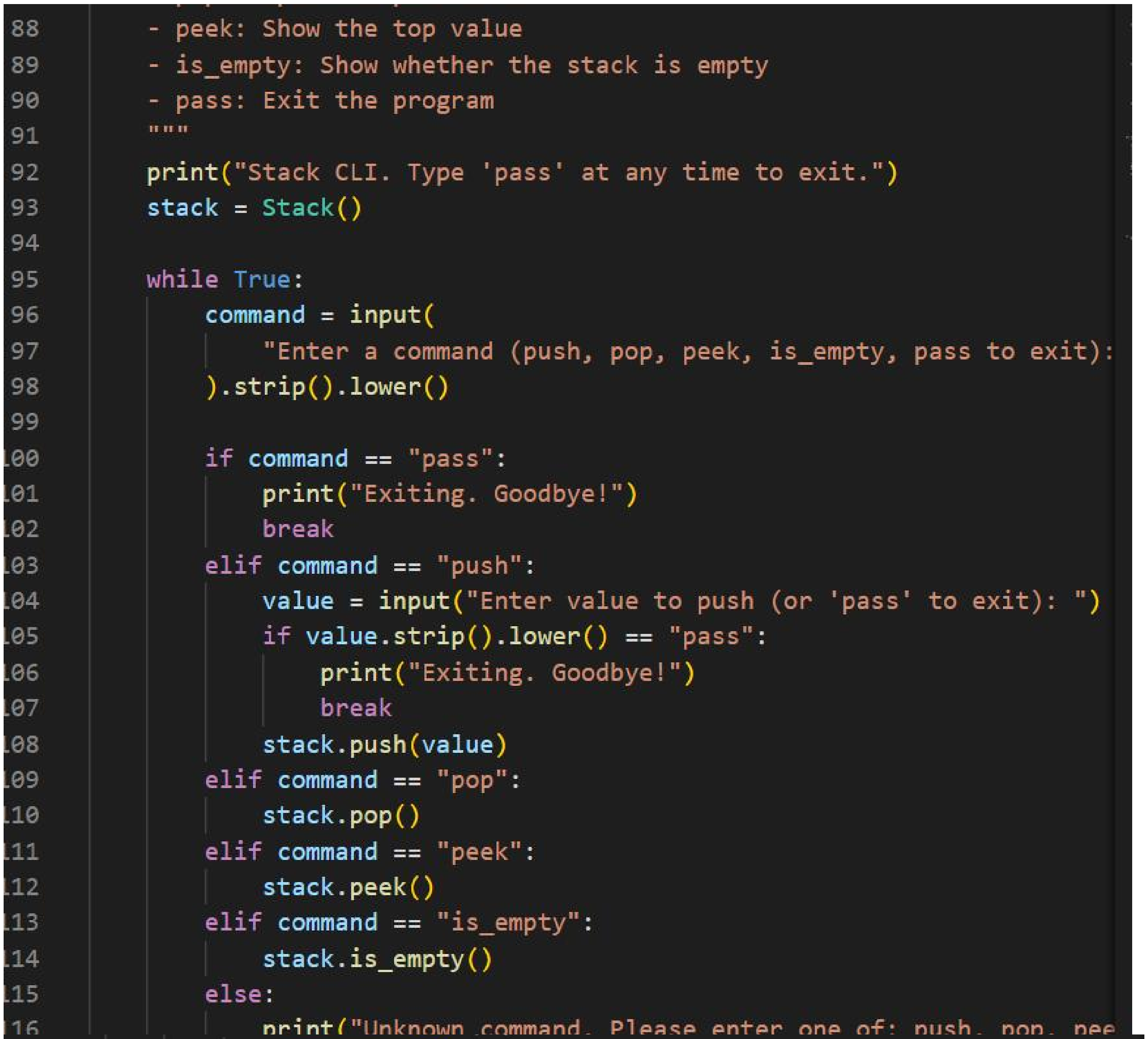
```python
class Stack:
    """A simple LIFO (last-in, first-out) stack implementation.

    Methods
    -------
    push(item):
        Pushes an item onto the top of the stack and prints the action.

    pop():
        Removes and returns the top item from the stack. Prints the act
        and handles underflow by printing a message and returning None.

    peek():
        Returns the top item without removing it. Prints the action and
        returns None if the stack is empty.

    is_empty():
        Returns True if the stack has no items; otherwise False. Also p
        the current empty state.
    """

    def __init__(self):
        """Initialize an empty stack."""
        self._items = []
        print("Initialized an empty stack.")

    def push(self, item):
        """Push an item onto the top of the stack.

        Parameters
        ----------
        item : Any
```



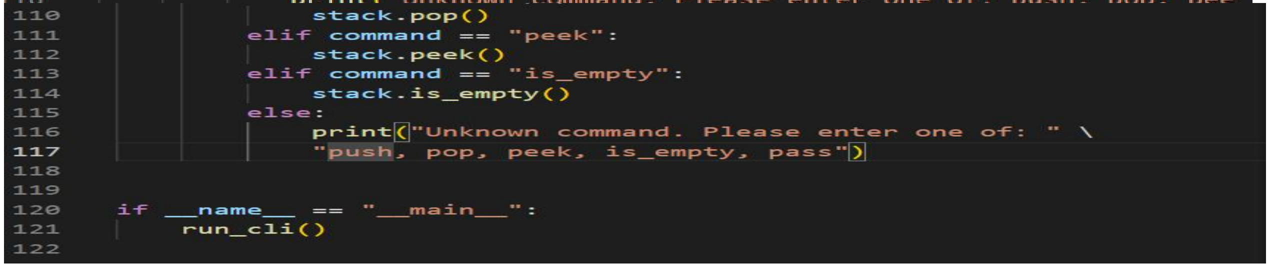
```
            The item to push onto the stack.
        """
        self._items.append(item)
        print(f"Pushed: {item}. Current stack: {self._items}")

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

        Returns
        -------
        Any | None
            The popped item, or None if the stack is empty.
        """
        if self.is_empty():
            print("Pop attempted on empty stack. Nothing to pop.")
            return None
        item = self._items.pop()
        print(f"Popped: {item}. Current stack: {self._items}")
        return item

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

        Returns
        -------
        Any | None
            The top item, or None if the stack is empty.
```



```
        if self.is_empty():
            print("Peek on empty stack. Top item: None")
            return None
        top_item = self._items[-1]
        print(f"Peeked: {top_item}. Current stack: {self._items}")
        return top_item

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

        Returns
        -------
        bool
            True if the stack is empty, else False.
        """
        empty = len(self._items) == 0
        print(f"Stack is empty: {empty}")
        return empty


def run_cli():
    """Run a simple command-line interface for interacting with the Sta

    Commands
    --------
    - push: Push a value (you will be prompted for the value)
    - pop: Pop the top value
```

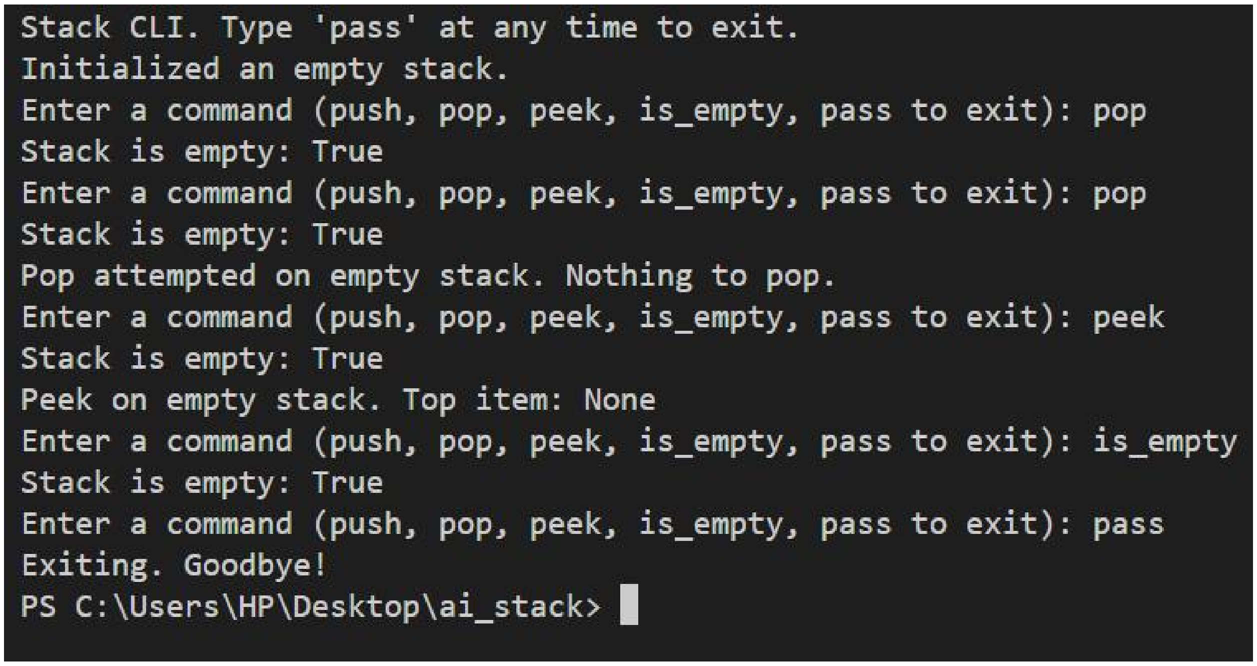


```python
    - peek: Show the top value
    - is_empty: Show whether the stack is empty
    - pass: Exit the program
    """
    print("Stack CLI. Type 'pass' at any time to exit.")
    stack = Stack()

    while True:
        command = input(
            "Enter a command (push, pop, peek, is_empty, pass to exit):
        ).strip().lower()

        if command == "pass":
            print("Exiting. Goodbye!")
            break
        elif command == "push":
            value = input("Enter value to push (or 'pass' to exit): ")
            if value.strip().lower() == "pass":
                print("Exiting. Goodbye!")
                break
            stack.push(value)
        elif command == "pop":
            stack.pop()
        elif command == "peek":
            stack.peek()
        elif command == "is_empty":
            stack.is_empty()
        else:
            print("Unknown command. Please enter one of: push, pop, pee
```



```python
            stack.pop()
        elif command == "peek":
            stack.peek()
        elif command == "is_empty":
            stack.is_empty()
        else:
            print("Unknown command. Please enter one of: " \
            "push, pop, peek, is_empty, pass")


if __name__ == "__main__":
    run_cli()
```

Output:



```
Stack CLI. Type 'pass' at any time to exit.
Initialized an empty stack.
Enter a command (push, pop, peek, is_empty, pass to exit): pop
Stack is empty: True
Enter a command (push, pop, peek, is_empty, pass to exit): pop
Stack is empty: True
Pop attempted on empty stack. Nothing to pop.
Enter a command (push, pop, peek, is_empty, pass to exit): peek
Stack is empty: True
Peek on empty stack. Top item: None
Enter a command (push, pop, peek, is_empty, pass to exit): is_empty
Stack is empty: True
Enter a command (push, pop, peek, is_empty, pass to exit): pass
Exiting. Goodbye!
PS C:\Users\HP\Desktop\ai_stack>
```

Observation:

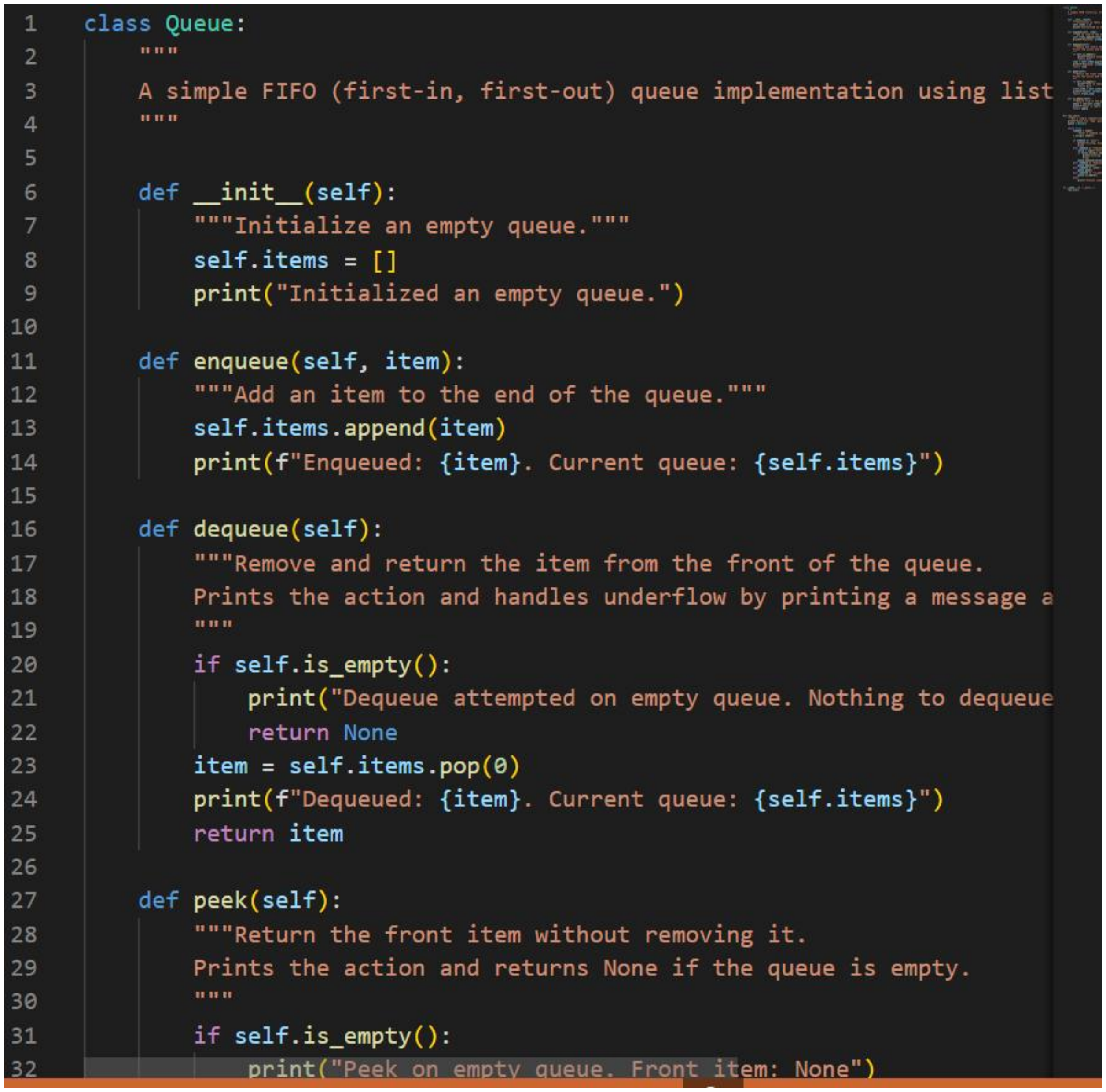
• Implemented a user-interactive LIFO Stack with push, pop, peek, and is_empty, including clear docstrings and state prints.
• CLI exits on typing "pass", ensuring simple, guided input and visible stack state after each action.

Task 2:

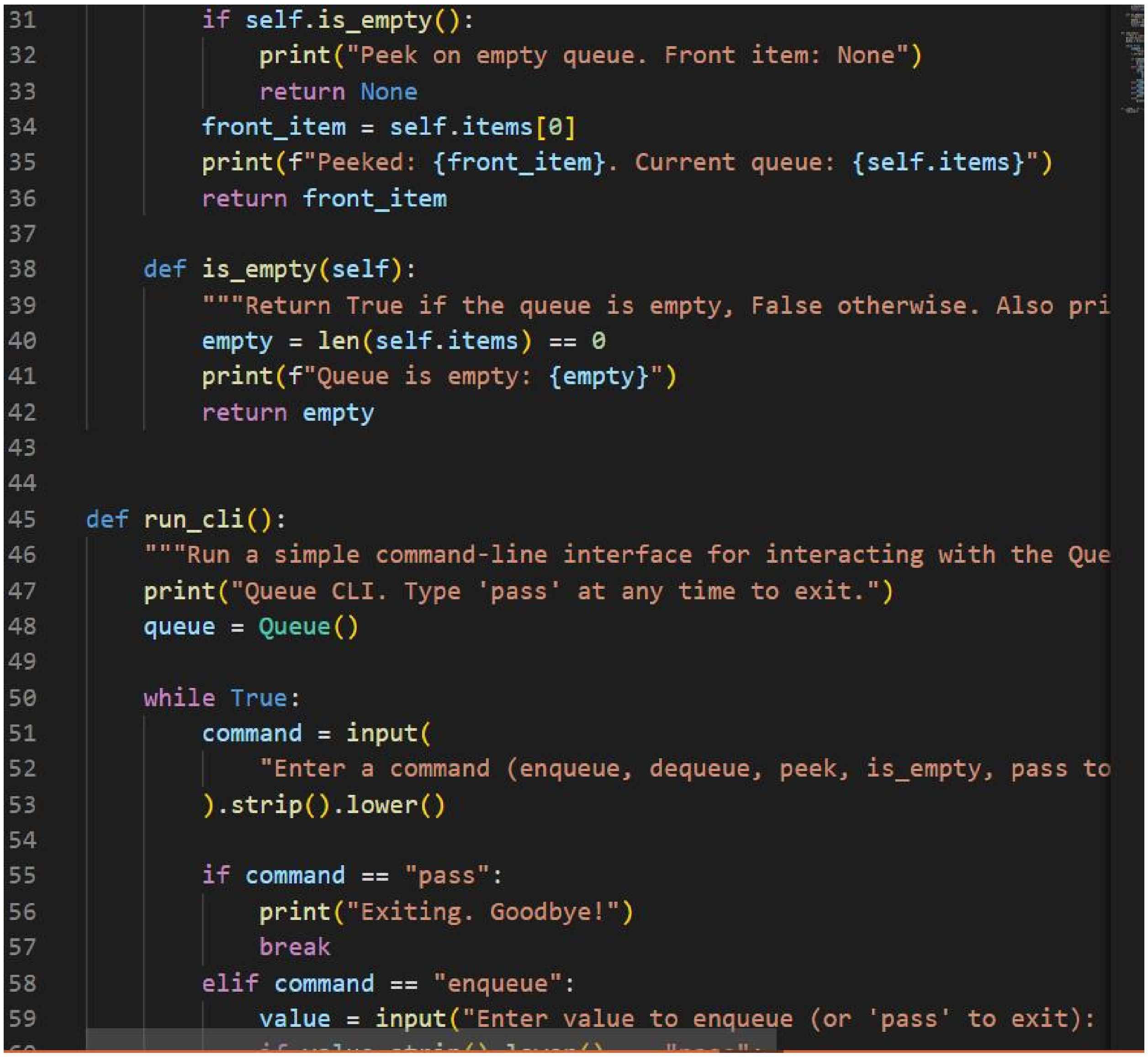
Prompt: Implement a Queue in Python using Lists

Use Python to implement a **FIFO (First-In-First-Out) Queue** using built-in **lists** (without using `collections.deque` or external libraries).

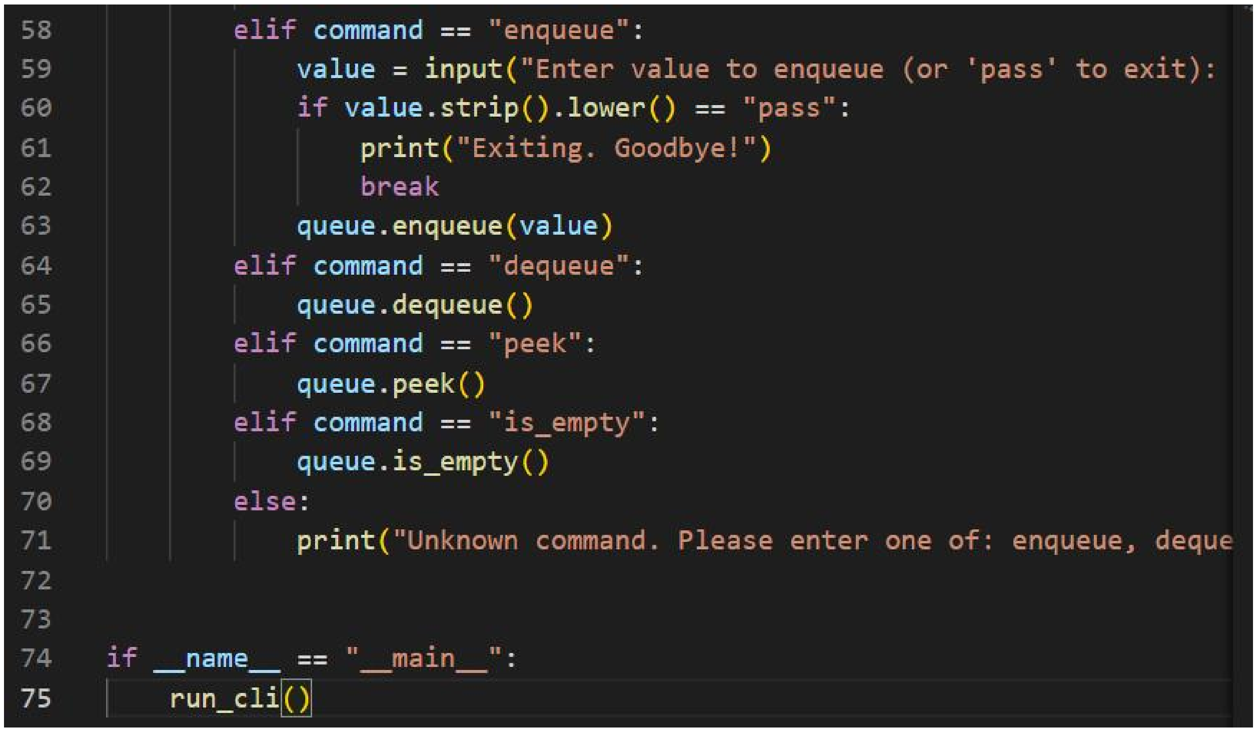
Code Generated:



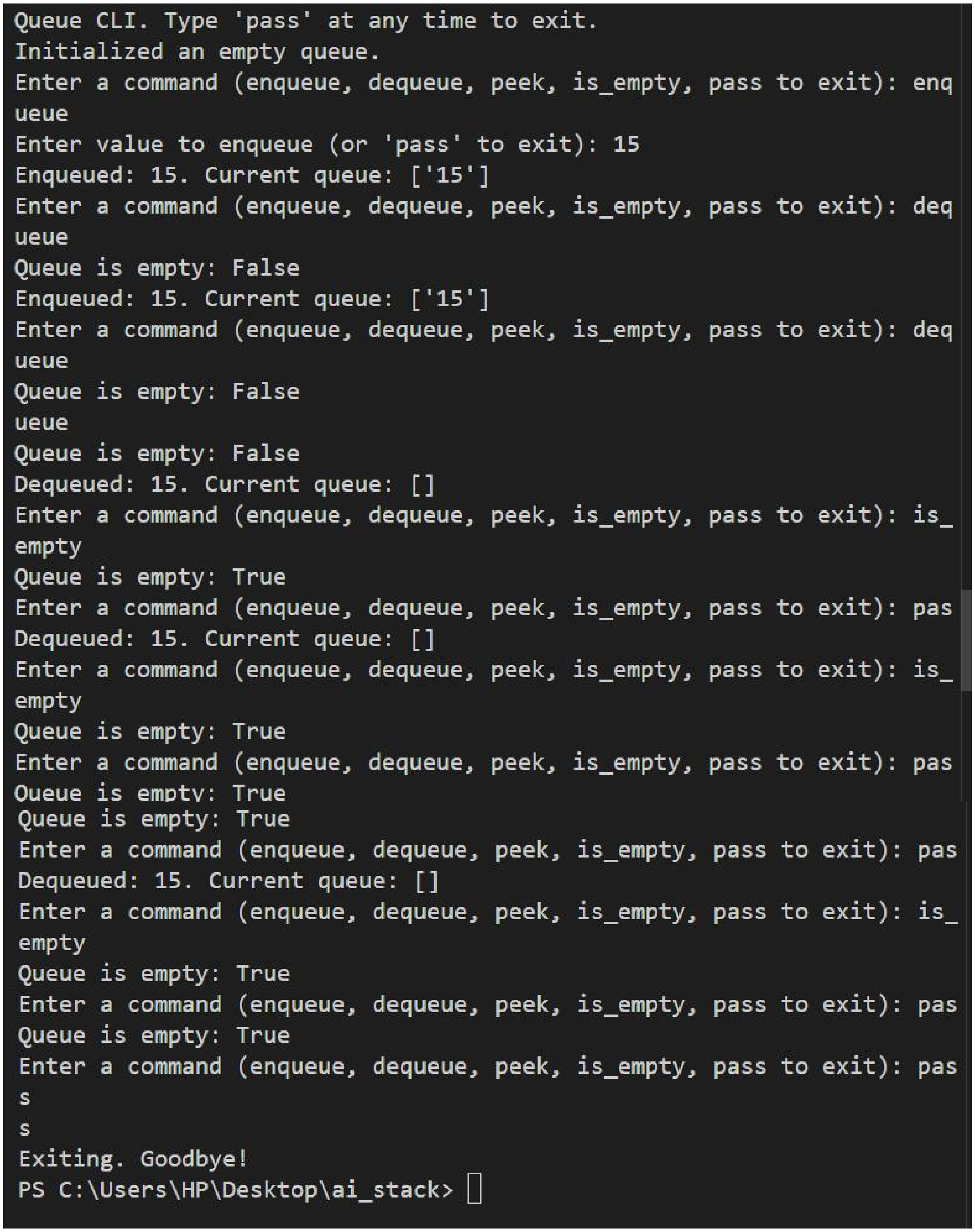
```python
class Queue:
    """
    A simple FIFO (first-in, first-out) queue implementation using list
    """

    def __init__(self):
        """Initialize an empty queue."""
        self.items = []
        print("Initialized an empty queue.")

    def enqueue(self, item):
        """Add an item to the end of the queue."""
        self.items.append(item)
        print(f"Enqueued: {item}. Current queue: {self.items}")

    def dequeue(self):
        """Remove and return the item from the front of the queue.
        Prints the action and handles underflow by printing a message a
        """
        if self.is_empty():
            print("Dequeue attempted on empty queue. Nothing to dequeue
            return None
        item = self.items.pop(0)
        print(f"Dequeued: {item}. Current queue: {self.items}")
        return item

    def peek(self):
        """Return the front item without removing it.
        Prints the action and returns None if the queue is empty.
        """
        if self.is_empty():
            print("Peek on empty queue. Front item: None")
```



```python
        if self.is_empty():
            print("Peek on empty queue. Front item: None")
            return None
        front_item = self.items[0]
        print(f"Peeked: {front_item}. Current queue: {self.items}")
        return front_item

    def is_empty(self):
        """Return True if the queue is empty, False otherwise. Also pri
        empty = len(self.items) == 0
        print(f"Queue is empty: {empty}")
        return empty


def run_cli():
    """Run a simple command-line interface for interacting with the Que
    print("Queue CLI. Type 'pass' at any time to exit.")
    queue = Queue()

    while True:
        command = input(
            "Enter a command (enqueue, dequeue, peek, is_empty, pass to
        ).strip().lower()

        if command == "pass":
            print("Exiting. Goodbye!")
            break
        elif command == "enqueue":
            value = input("Enter value to enqueue (or 'pass' to exit):
```


```
        elif command == "enqueue":
            value = input("Enter value to enqueue (or 'pass' to exit):
            if value.strip().lower() == "pass":
                print("Exiting. Goodbye!")
                break
            queue.enqueue(value)
        elif command == "dequeue":
            queue.dequeue()
        elif command == "peek":
            queue.peek()
        elif command == "is_empty":
            queue.is_empty()
        else:
            print("Unknown command. Please enter one of: enqueue, deque


if __name__ == "__main__":
    run_cli()
```

Output:



```
Queue CLI. Type 'pass' at any time to exit.
Initialized an empty queue.
Enter a command (enqueue, dequeue, peek, is_empty, pass to exit): enq
ueue
Enter value to enqueue (or 'pass' to exit): 15
Enqueued: 15. Current queue: ['15']
Enter a command (enqueue, dequeue, peek, is_empty, pass to exit): deq
ueue
Queue is empty: False
Enqueued: 15. Current queue: ['15']
Enter a command (enqueue, dequeue, peek, is_empty, pass to exit): deq
ueue
Queue is empty: False
ueue
Queue is empty: False
Dequeued: 15. Current queue: []
Enter a command (enqueue, dequeue, peek, is_empty, pass to exit): is_
empty
Queue is empty: True
Enter a command (enqueue, dequeue, peek, is_empty, pass to exit): pas
Dequeued: 15. Current queue: []
Enter a command (enqueue, dequeue, peek, is_empty, pass to exit): is_
empty
Queue is empty: True
Enter a command (enqueue, dequeue, peek, is_empty, pass to exit): pas
Queue is empty: True
Queue is empty: True
Enter a command (enqueue, dequeue, peek, is_empty, pass to exit): pas
Dequeued: 15. Current queue: []
Enter a command (enqueue, dequeue, peek, is_empty, pass to exit): is_
empty
Queue is empty: True
Enter a command (enqueue, dequeue, peek, is_empty, pass to exit): pas
Queue is empty: True
Enter a command (enqueue, dequeue, peek, is_empty, pass to exit): pas
s
s
Exiting. Goodbye!
PS C:\Users\HP\Desktop\ai_stack>
```

Observation:

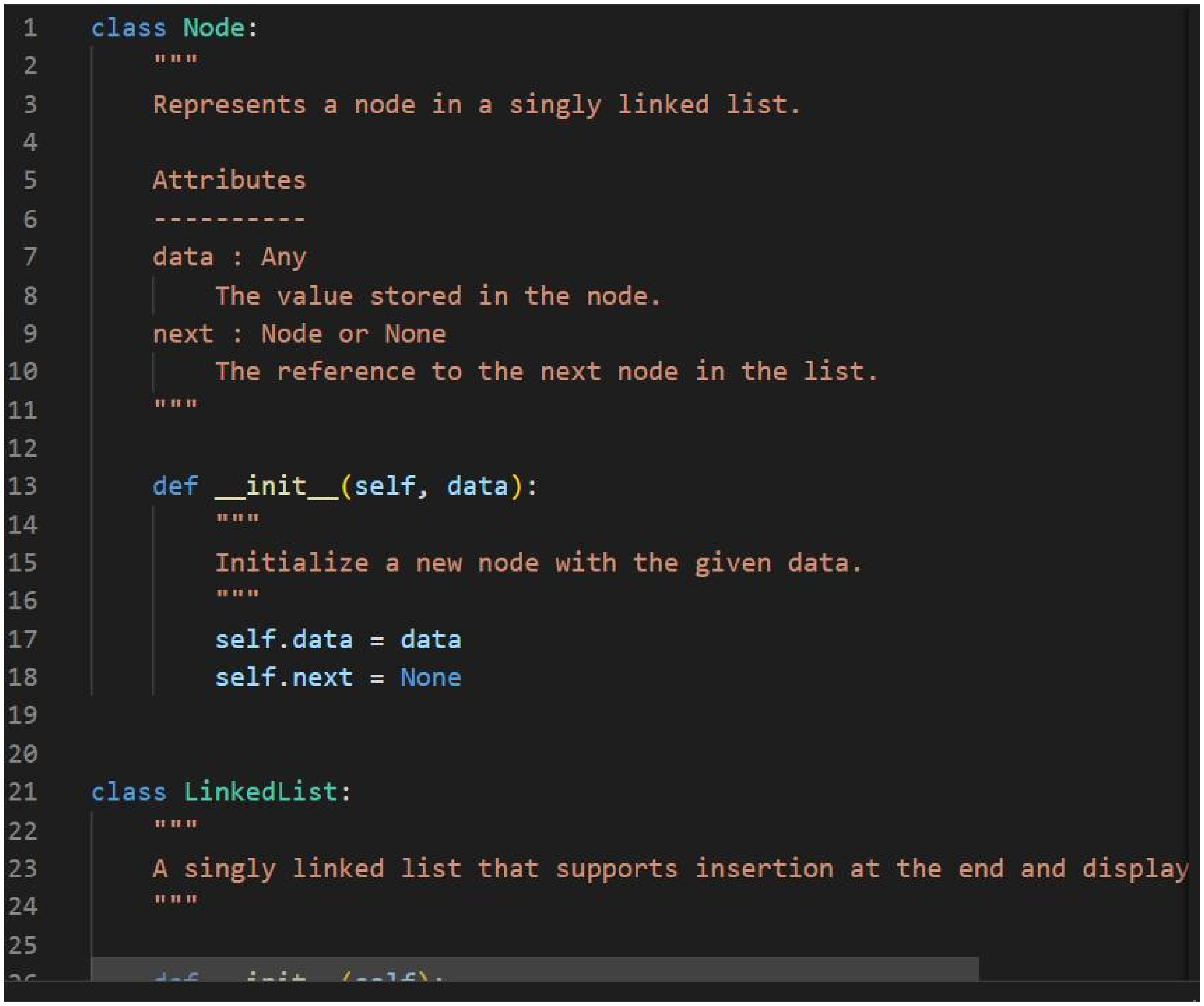
The provided Queue implementation uses Python lists to achieve FIFO behavior, includes all essential methods (enqueue, dequeue, peek, is_empty), and features print statements for user feedback. The code also offers a command-line interface for interactive queue operations, making it practical and easy to use.

Task3:

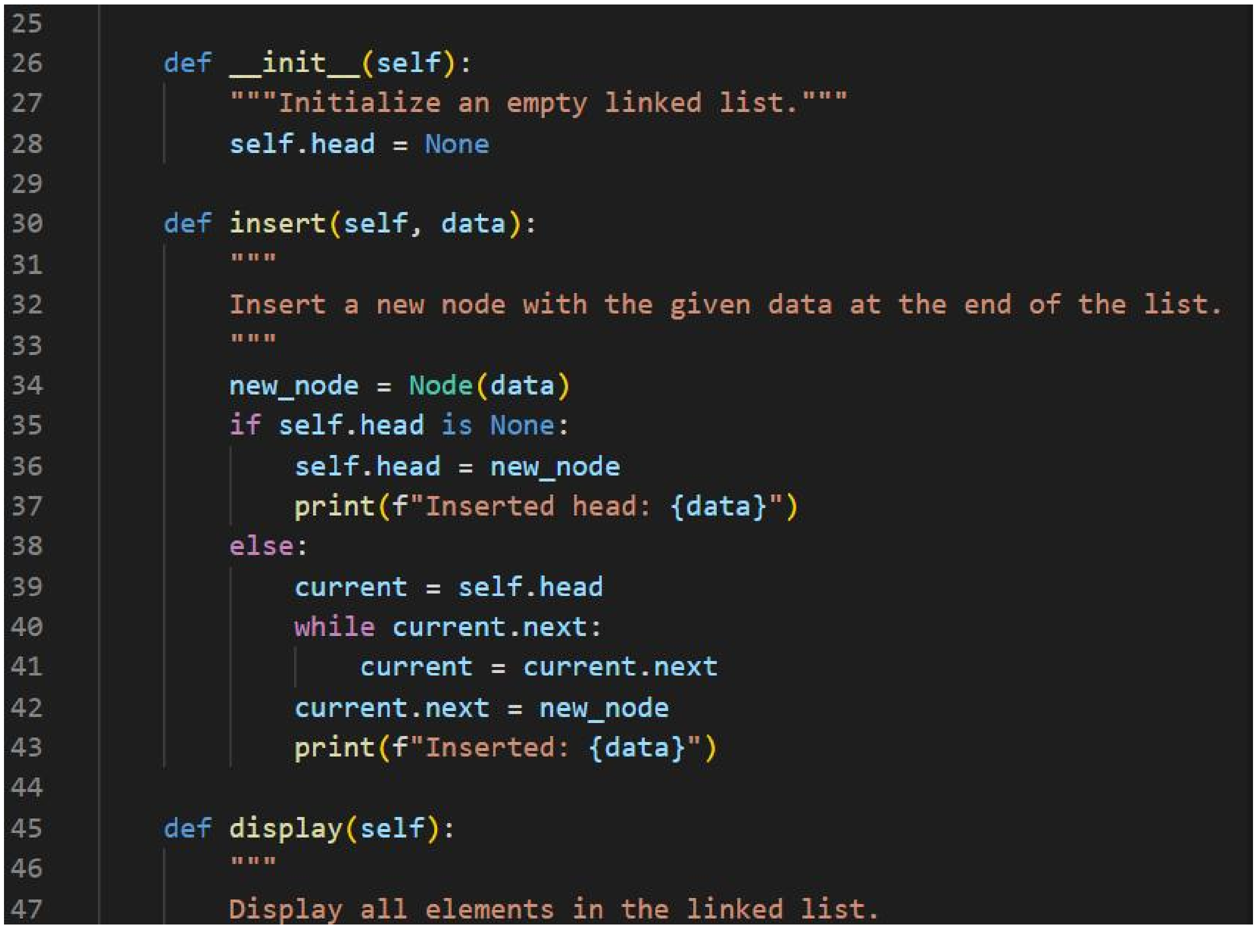
Prompt:

"Generate a Python implementation of a singly linked list. The list should have a Node class and a LinkedList class. The LinkedList class must include methods to insert a new node at the end and display all elements. Each method should have clear docstrings explaining its purpose and usage."

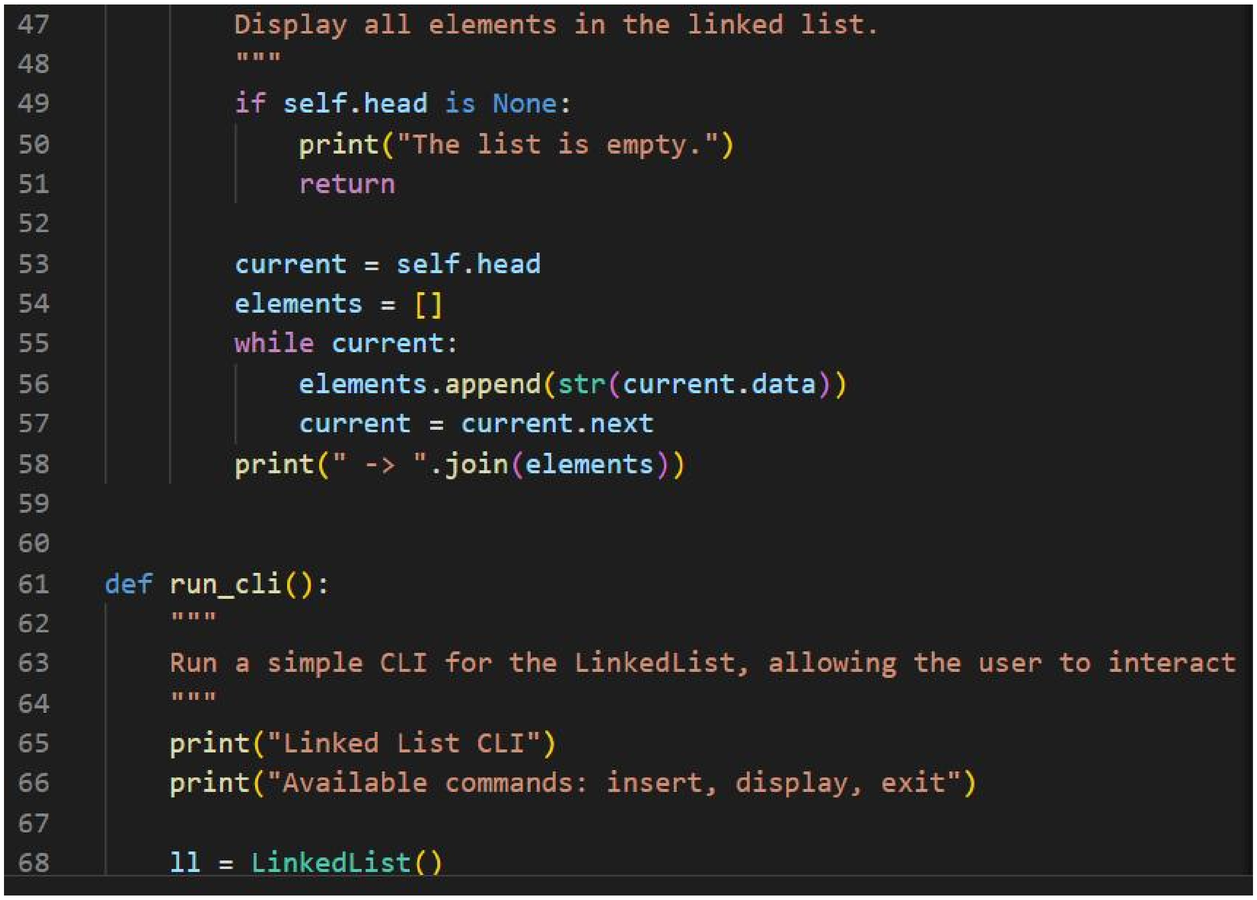
Code Generated:

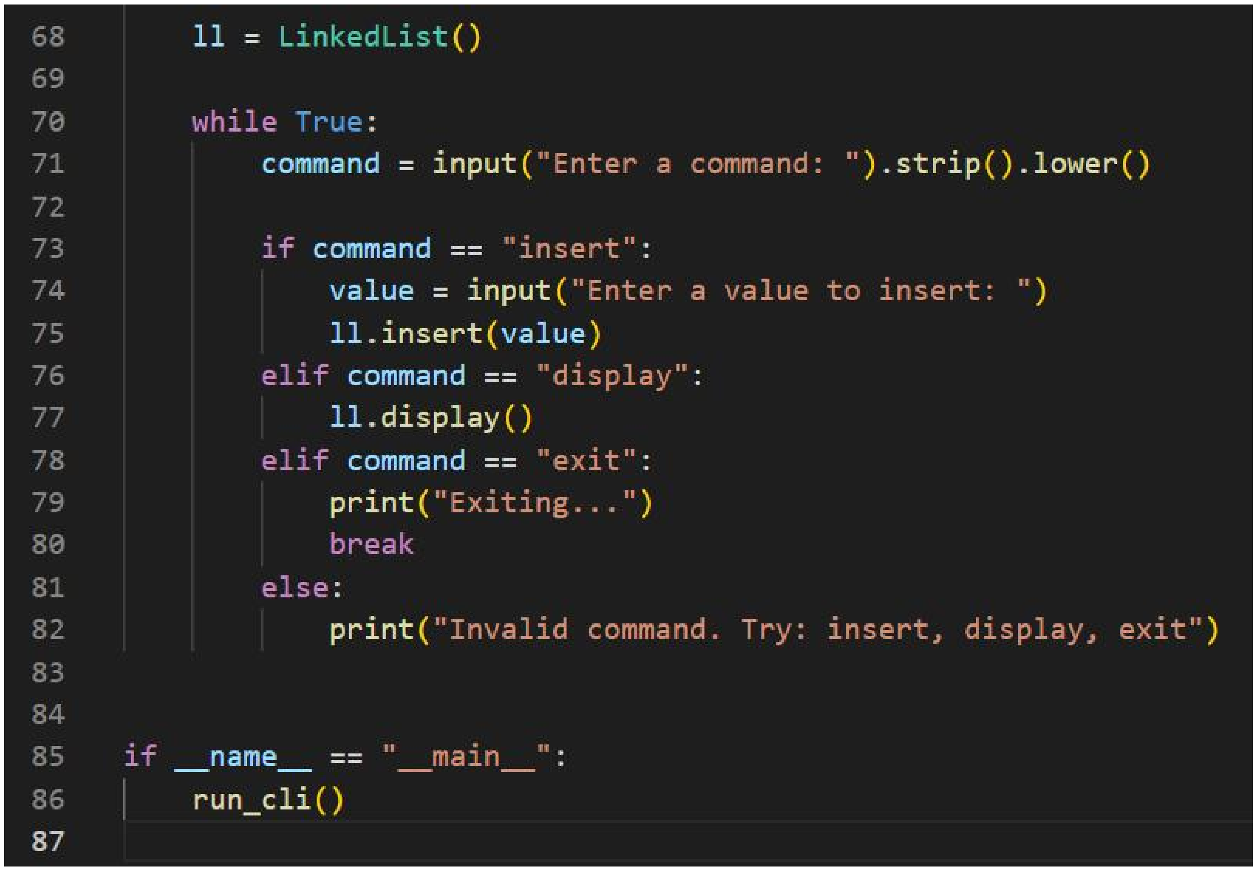


```python
class Node:
    """
    Represents a node in a singly linked list.

    Attributes
    ----------
    data : Any
        The value stored in the node.
    next : Node or None
        The reference to the next node in the list.
    """

    def __init__(self, data):
        """
        Initialize a new node with the given data.
        """
        self.data = data
        self.next = None


class LinkedList:
    """
    A singly linked list that supports insertion at the end and display
    """

    def __init__(self):
```



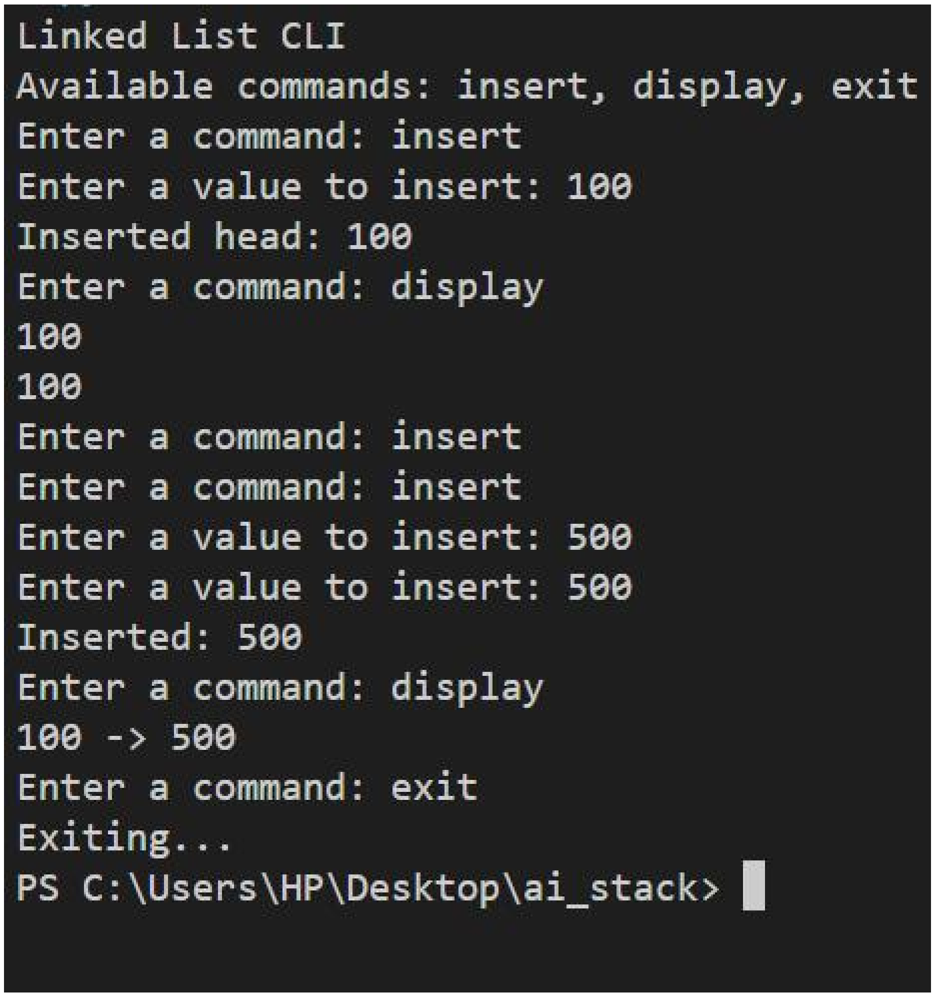
```

    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None

    def insert(self, data):
        """
        Insert a new node with the given data at the end of the list.
        """
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            print(f"Inserted head: {data}")
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
            print(f"Inserted: {data}")

    def display(self):
        """
        Display all elements in the linked list.
```


```
        Display all elements in the linked list.
        """
        if self.head is None:
            print("The list is empty.")
            return

        current = self.head
        elements = []
        while current:
            elements.append(str(current.data))
            current = current.next
        print(" -> ".join(elements))


def run_cli():
    """
    Run a simple CLI for the LinkedList, allowing the user to interact
    """
    print("Linked List CLI")
    print("Available commands: insert, display, exit")

    ll = LinkedList()
```



```python
    ll = LinkedList()

    while True:
        command = input("Enter a command: ").strip().lower()

        if command == "insert":
            value = input("Enter a value to insert: ")
            ll.insert(value)
        elif command == "display":
            ll.display()
        elif command == "exit":
            print("Exiting...")
            break
        else:
            print("Invalid command. Try: insert, display, exit")


if __name__ == "__main__":
    run_cli()

```

Output:



```
Linked List CLI
Available commands: insert, display, exit
Enter a command: insert
Enter a value to insert: 100
Inserted head: 100
Enter a command: display
100
100
Enter a command: insert
Enter a command: insert
Enter a value to insert: 500
Enter a value to insert: 500
Inserted: 500
Enter a command: display
100 -> 500
Enter a command: exit
Exiting...
PS C:\Users\HP\Desktop\ai_stack>
```

Observation:

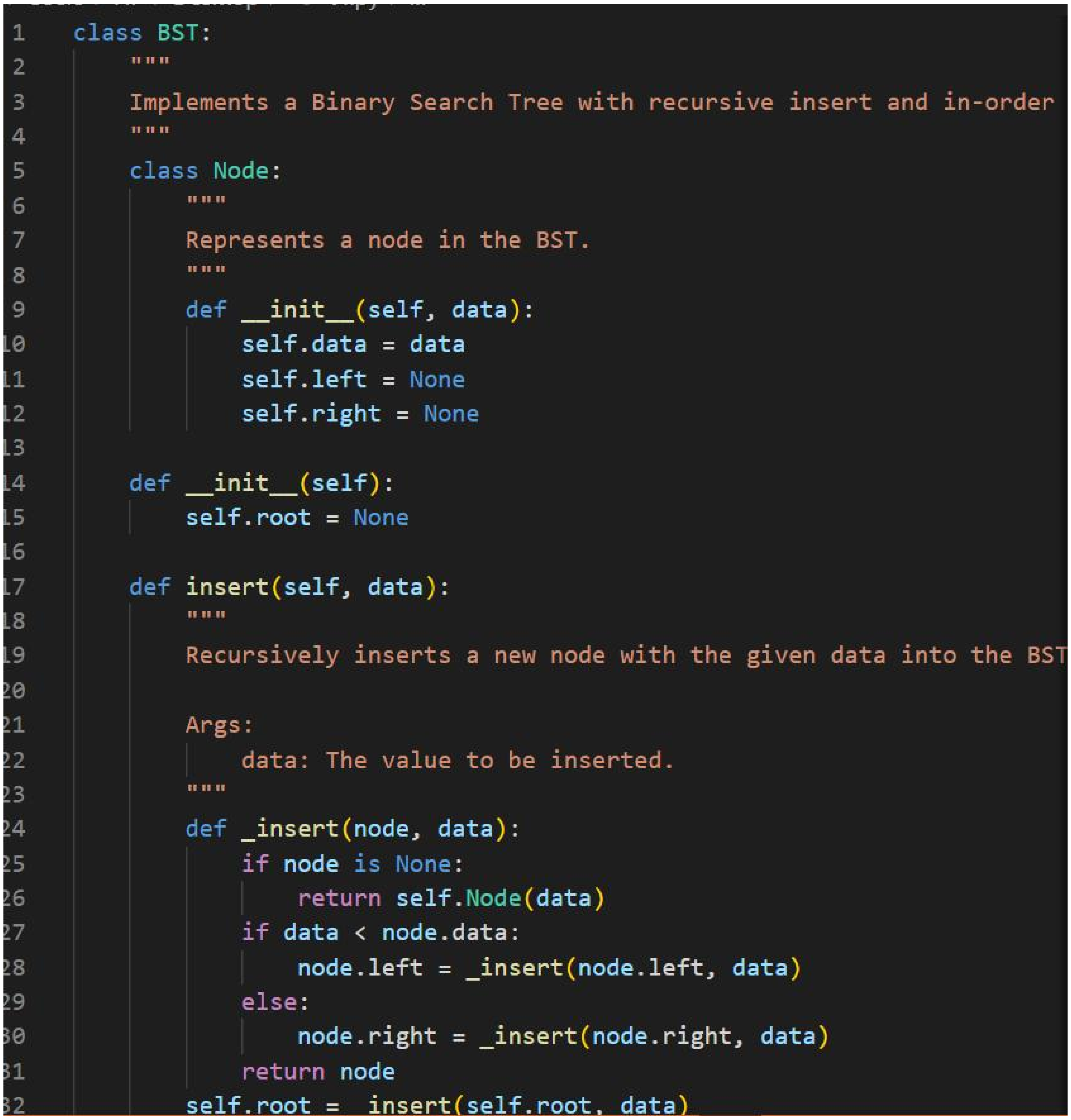
The provided singly linked list implementation in Python uses two classes: Node and LinkedList. The Node class stores individual data and a reference to the next node, while the LinkedList class manages the list, allowing insertion of new nodes at the end and displaying all elements. Each method is clearly documented, making the code easy to understand and maintain. This structure is suitable for basic linked list operations and can be extended for additional functionality if needed.

Task4:

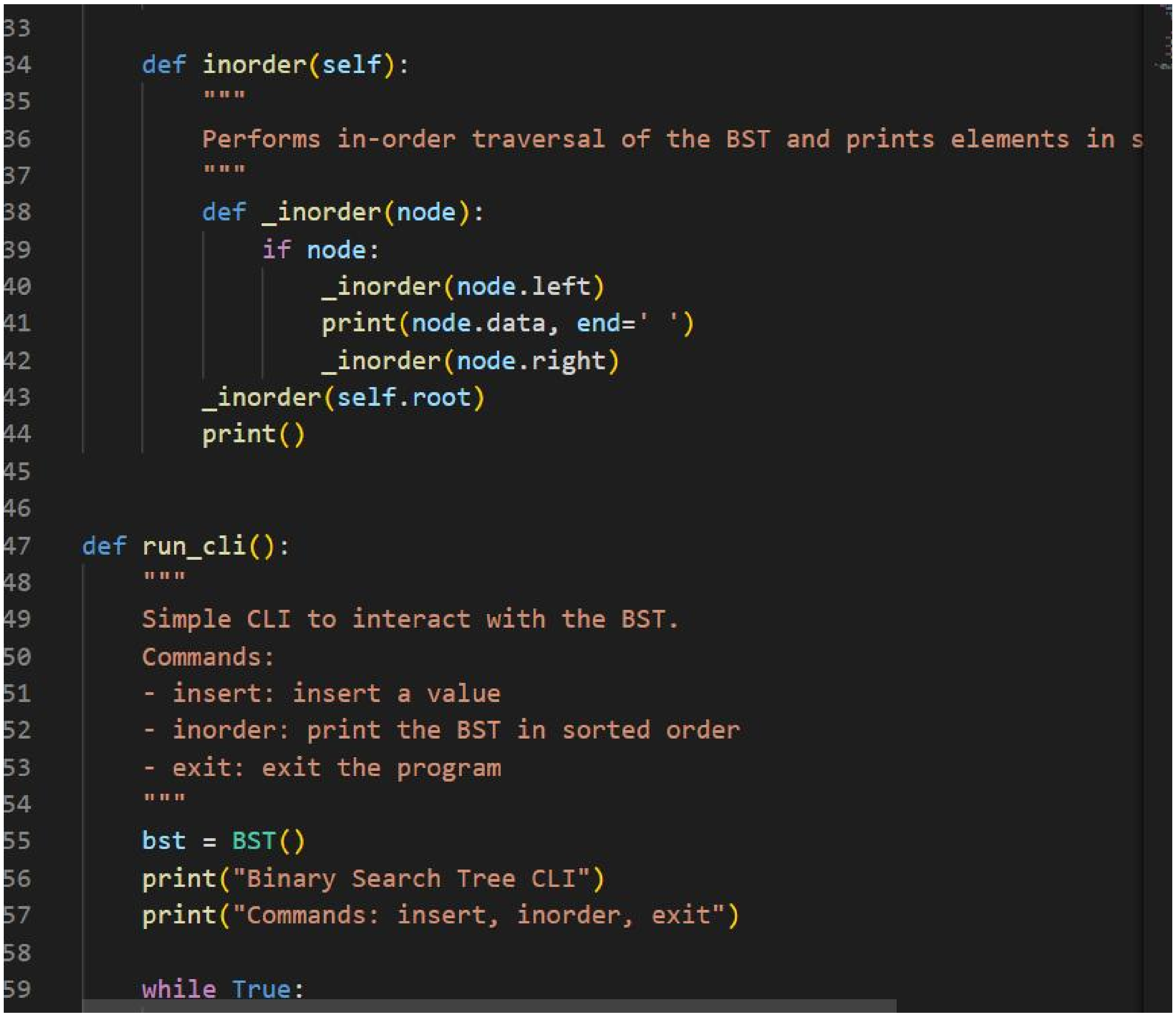
Prompt: Binary Search Tree (BST)

"Generate a Python implementation of a Binary Search Tree (BST) class. The BST should support recursive insertion of nodes and provide an in-order traversal method that prints the elements in sorted order. Include clear docstrings for each method."

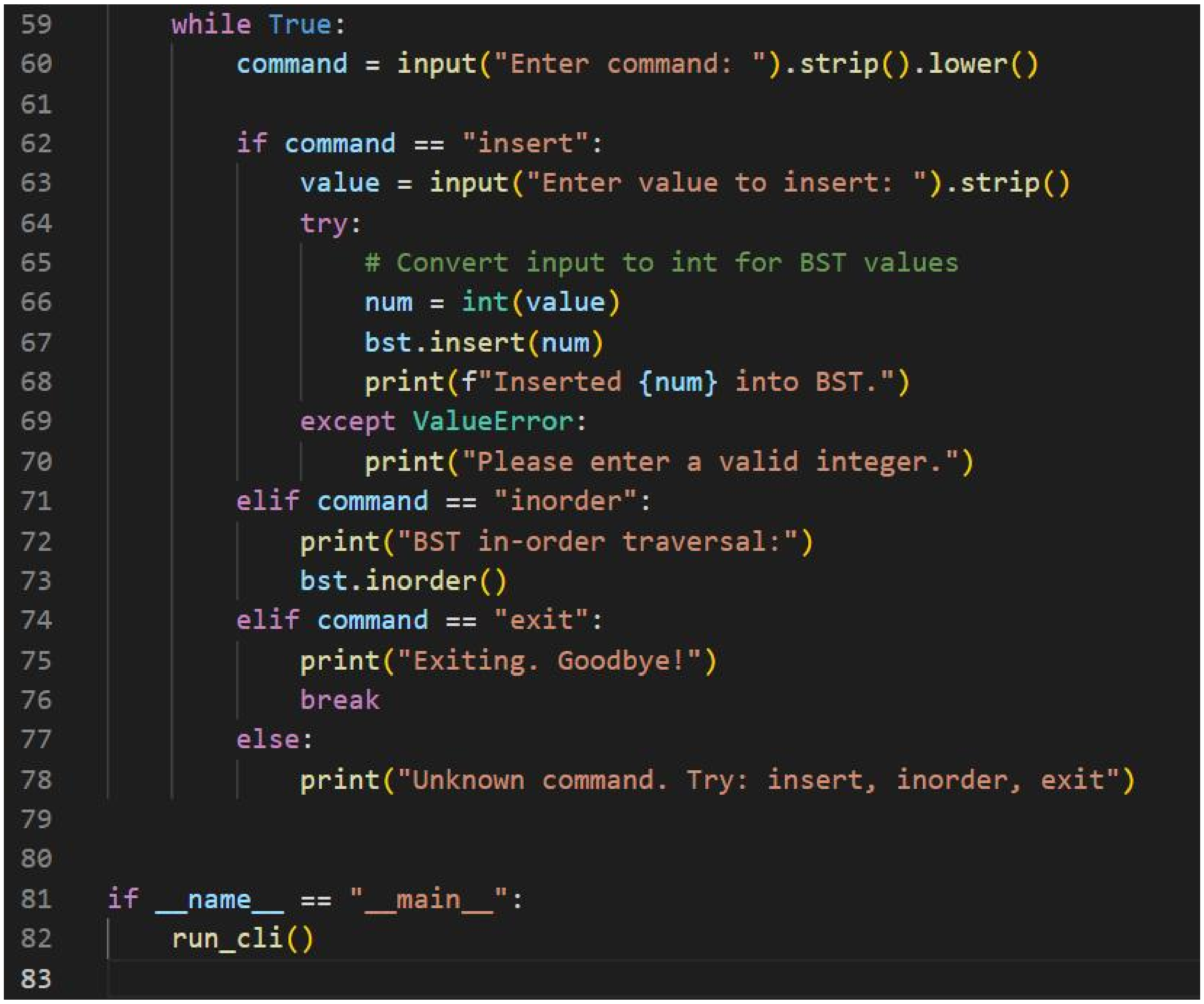
Code Generated:



```python
class BST:
    """
    Implements a Binary Search Tree with recursive insert and in-order
    """
    class Node:
        """
        Represents a node in the BST.
        """
        def __init__(self, data):
            self.data = data
            self.left = None
            self.right = None

    def __init__(self):
        self.root = None

    def insert(self, data):
        """
        Recursively inserts a new node with the given data into the BST

        Args:
            data: The value to be inserted.
        """
        def _insert(node, data):
            if node is None:
                return self.Node(data)
            if data < node.data:
                node.left = _insert(node.left, data)
            else:
                node.right = _insert(node.right, data)
            return node
        self.root = _insert(self.root, data)
```

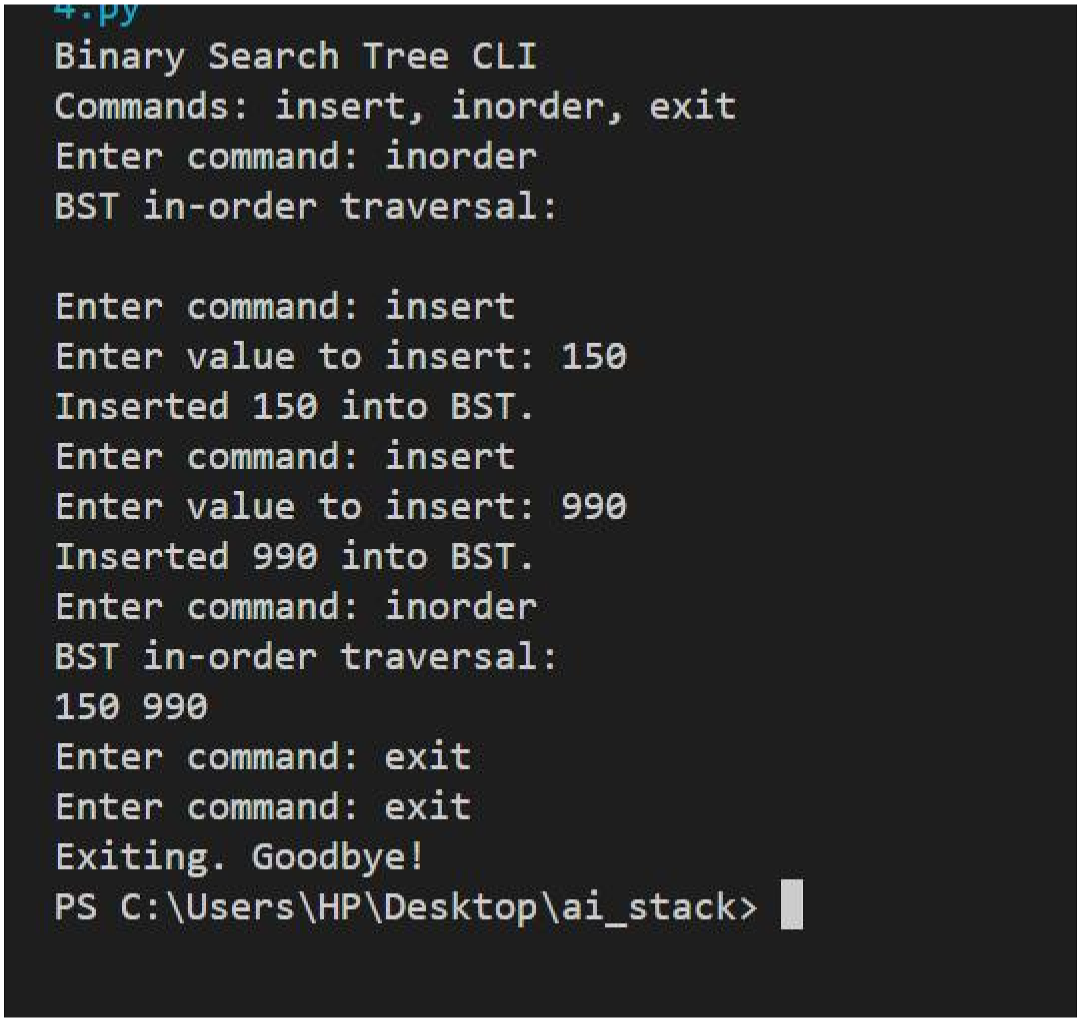


```
    def inorder(self):
        """
        Performs in-order traversal of the BST and prints elements in s
        """
        def _inorder(node):
            if node:
                _inorder(node.left)
                print(node.data, end=' ')
                _inorder(node.right)
        _inorder(self.root)
        print()


def run_cli():
    """
    Simple CLI to interact with the BST.
    Commands:
    - insert: insert a value
    - inorder: print the BST in sorted order
    - exit: exit the program
    """
    bst = BST()
    print("Binary Search Tree CLI")
    print("Commands: insert, inorder, exit")

    while True:
```



```python
    while True:
        command = input("Enter command: ").strip().lower()

        if command == "insert":
            value = input("Enter value to insert: ").strip()
            try:
                # Convert input to int for BST values
                num = int(value)
                bst.insert(num)
                print(f"Inserted {num} into BST.")
            except ValueError:
                print("Please enter a valid integer.")
        elif command == "inorder":
            print("BST in-order traversal:")
            bst.inorder()
        elif command == "exit":
            print("Exiting. Goodbye!")
            break
        else:
            print("Unknown command. Try: insert, inorder, exit")


if __name__ == "__main__":
    run_cli()

```

Output:



```
4.py
Binary Search Tree CLI
Commands: insert, inorder, exit
Enter command: inorder
BST in-order traversal:

Enter command: insert
Enter value to insert: 150
Inserted 150 into BST.
Enter command: insert
Enter value to insert: 990
Inserted 990 into BST.
Enter command: inorder
BST in-order traversal:
150 990
Enter command: exit
Enter command: exit
Exiting. Goodbye!
PS C:\Users\HP\Desktop\ai_stack>
```

Observation:

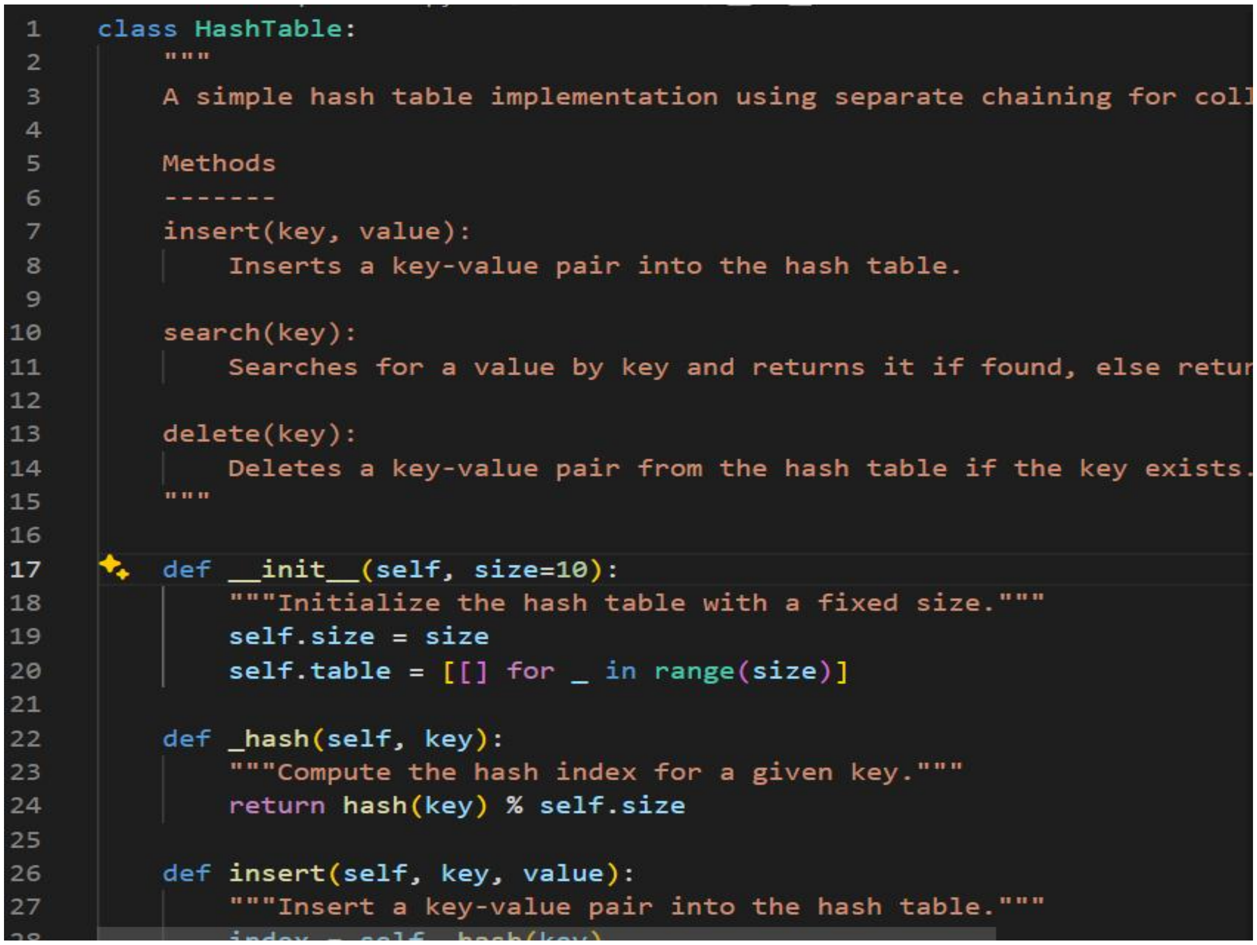
- The program lets users **insert numbers** into the binary search tree one by one.
- When the user types **`inorder`**, it prints all the numbers in sorted order.
- The program checks if the input for insertion is a valid integer and gives an error if not.
- Users can keep inserting or printing until they type **`exit`** to stop.
- Overall, it's easy to use and correctly shows how a BST works with user input.

Task5:

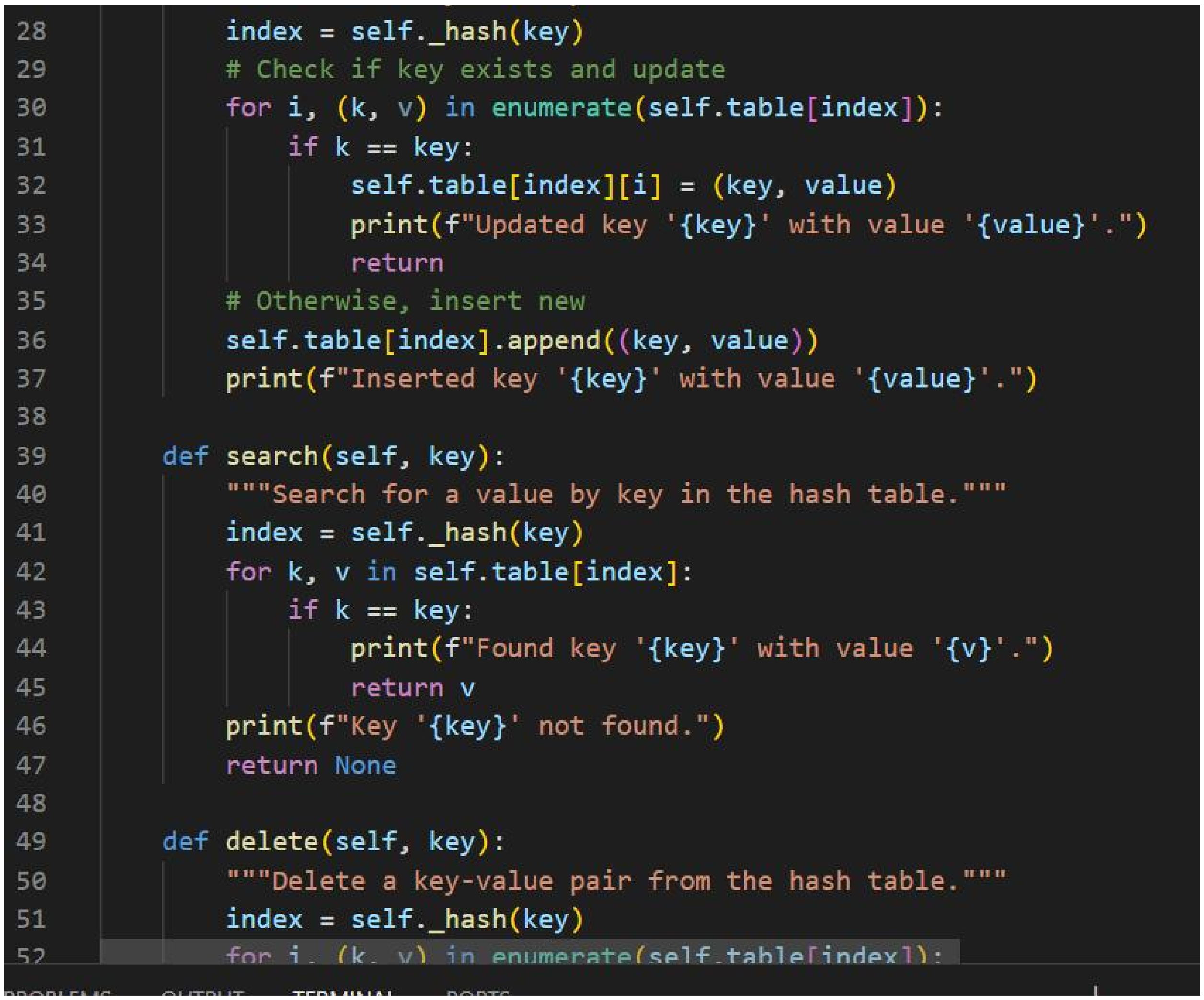
Prompt:

Implement a hash table with basic insert, search, and delete methods.
Sample Input Code:
class HashTable:
pass

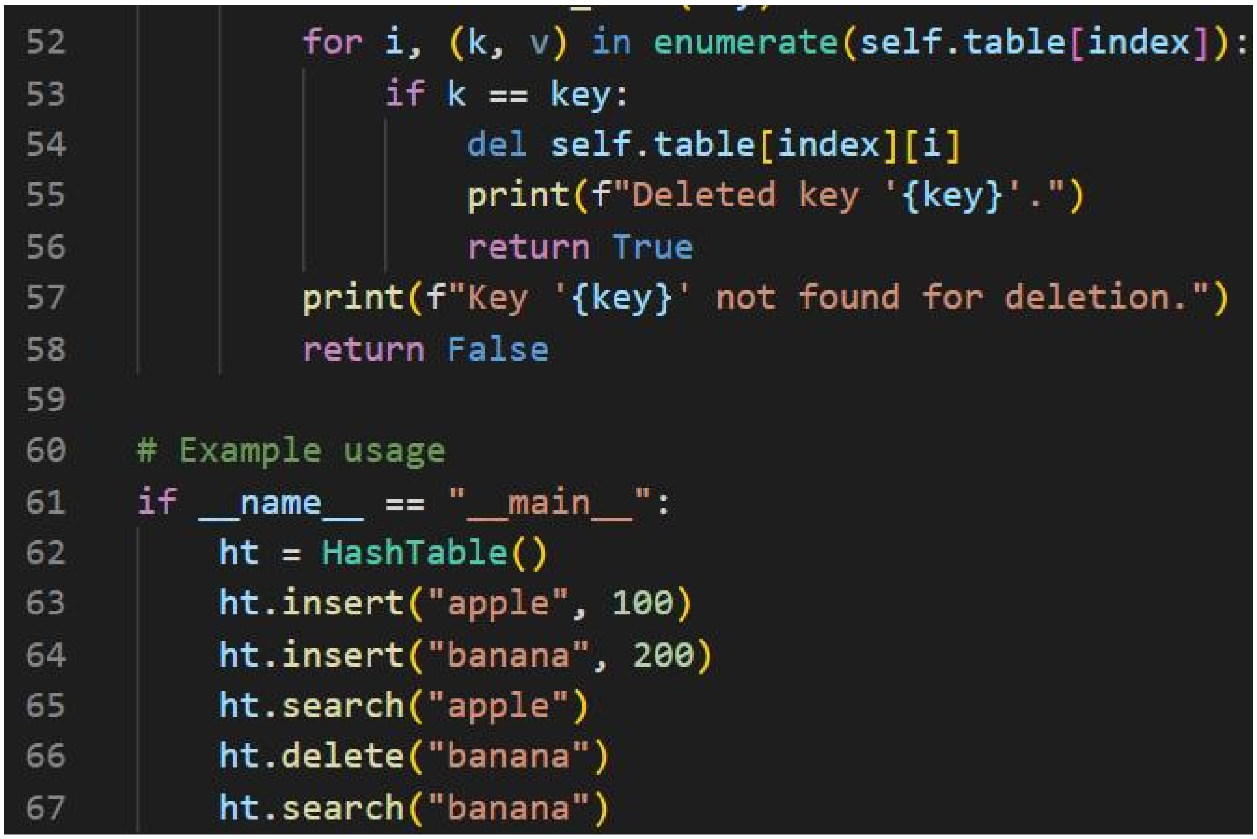
Code Generated:



```python
class HashTable:
    """
    A simple hash table implementation using separate chaining for coll

    Methods
    -------
    insert(key, value):
        Inserts a key-value pair into the hash table.

    search(key):
        Searches for a value by key and returns it if found, else retur

    delete(key):
        Deletes a key-value pair from the hash table if the key exists.
    """

    def __init__(self, size=10):
        """Initialize the hash table with a fixed size."""
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        """Compute the hash index for a given key."""
        return hash(key) % self.size

    def insert(self, key, value):
        """Insert a key-value pair into the hash table."""
```

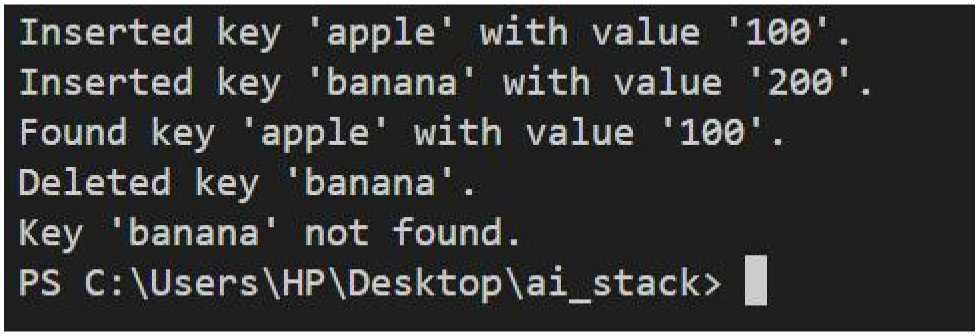


```python
        index = self._hash(key)
        # Check if key exists and update
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                print(f"Updated key '{key}' with value '{value}'.")
                return
        # Otherwise, insert new
        self.table[index].append((key, value))
        print(f"Inserted key '{key}' with value '{value}'.")

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

    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
```



```
            for i, (k, v) in enumerate(self.table[index]):
                if k == key:
                    del self.table[index][i]
                    print(f"Deleted key '{key}'.")
                    return True
            print(f"Key '{key}' not found for deletion.")
            return False

# Example usage
if __name__ == "__main__":
    ht = HashTable()
    ht.insert("apple", 100)
    ht.insert("banana", 200)
    ht.search("apple")
    ht.delete("banana")
    ht.search("banana")
```

Output:



```
Inserted key 'apple' with value '100'.
Inserted key 'banana' with value '200'.
Found key 'apple' with value '100'.
Deleted key 'banana'.
Key 'banana' not found.
PS C:\Users\HP\Desktop\ai_stack>
```

Observation:

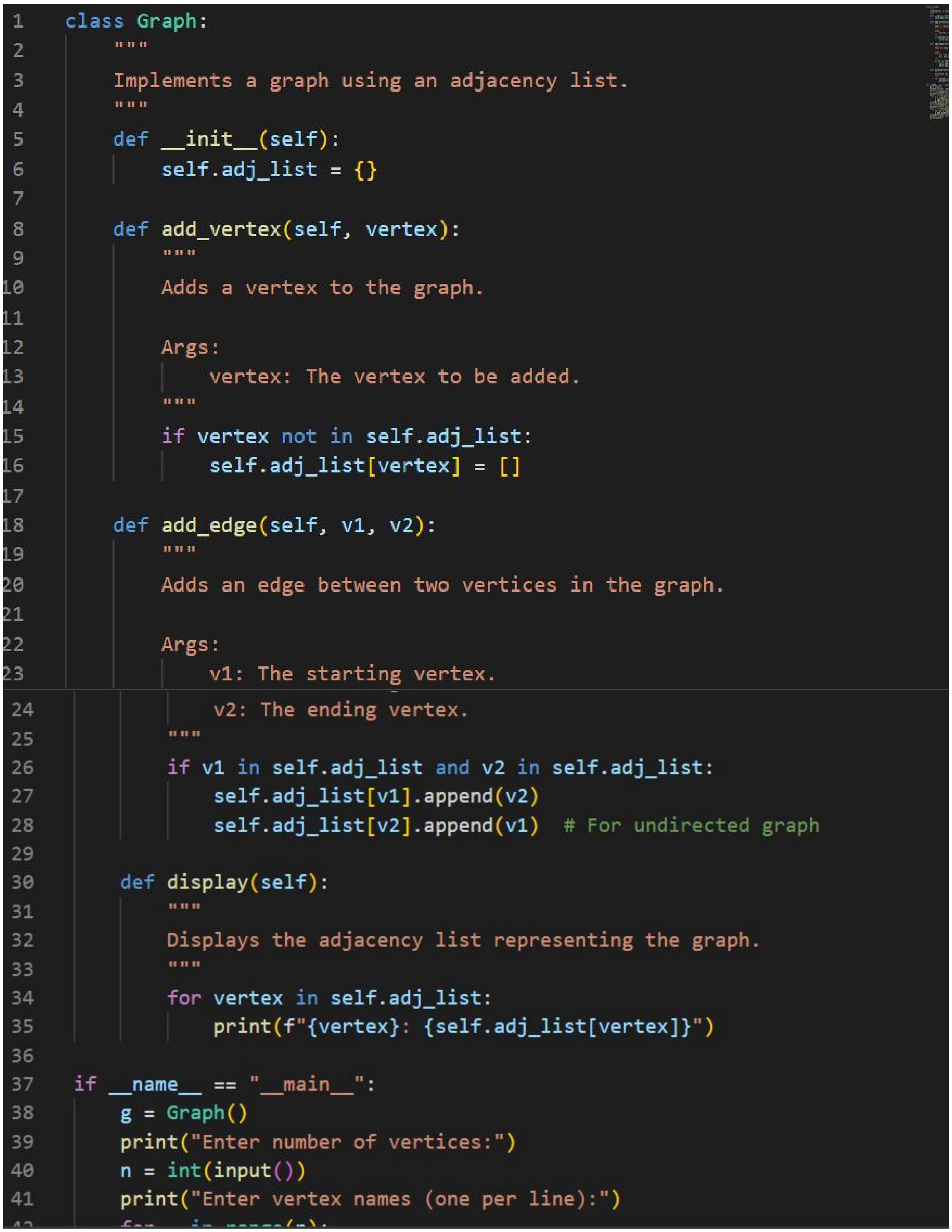
- The code implements a Hash Table using separate chaining (lists at each index) to handle collisions.
- It provides methods to insert, search, and delete key-value pairs efficiently.
- Keys are hashed using Python’s built-in hash() function and mapped within the fixed table size.
- Example usage shows correct behavior: inserting keys, updating, finding values, and handling deletion properly

Task6:

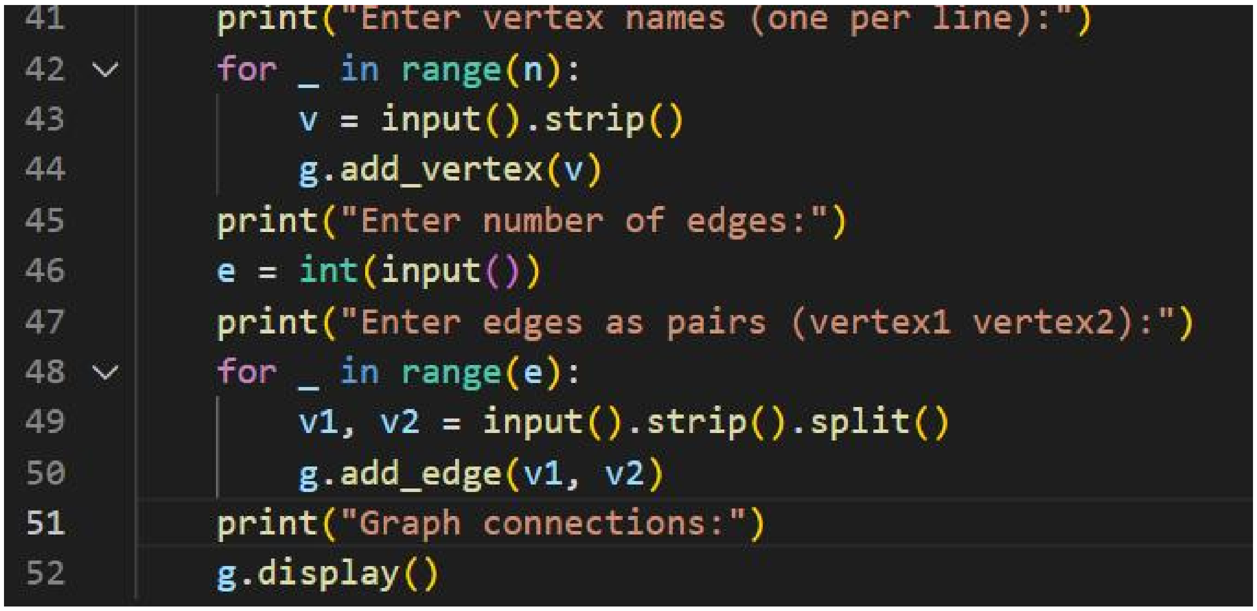
Prompt:

"Generate a Python implementation of a Graph class using an adjacency list. The class should include methods to add vertices, add edges, and display the connections. Each method should have clear docstrings explaining its purpose and usage."

Code Generated:

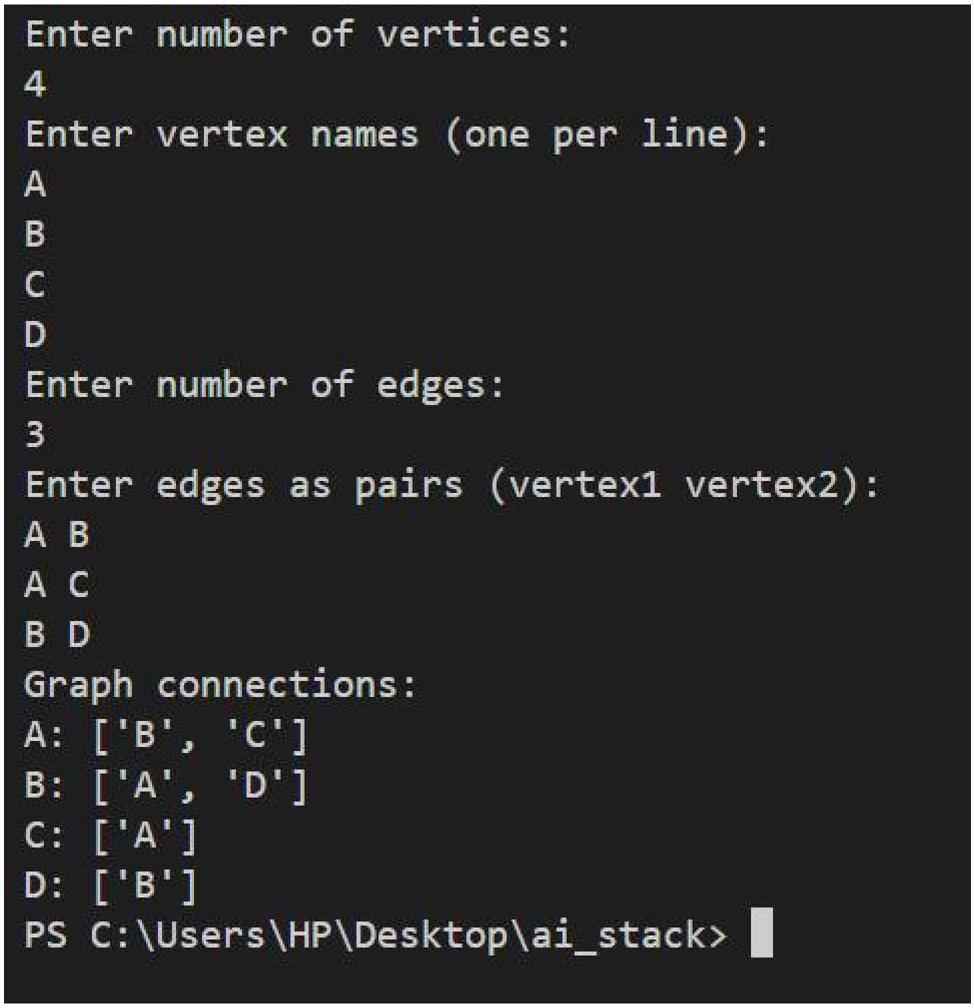


```python
class Graph:
    """
    Implements a graph using an adjacency list.
    """
    def __init__(self):
        self.adj_list = {}

    def add_vertex(self, vertex):
        """
        Adds a vertex to the graph.

        Args:
            vertex: The vertex to be added.
        """
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []

    def add_edge(self, v1, v2):
        """
        Adds an edge between two vertices in the graph.

        Args:
            v1: The starting vertex.
            v2: The ending vertex.
        """
        if v1 in self.adj_list and v2 in self.adj_list:
            self.adj_list[v1].append(v2)
            self.adj_list[v2].append(v1)  # For undirected graph

    def display(self):
        """
        Displays the adjacency list representing the graph.
        """
        for vertex in self.adj_list:
            print(f"{vertex}: {self.adj_list[vertex]}")

if __name__ == "__main__":
    g = Graph()
    print("Enter number of vertices:")
    n = int(input())
    print("Enter vertex names (one per line):")
    for _ in range(n):
```



```python
    print("Enter vertex names (one per line):")
    for _ in range(n):
        v = input().strip()
        g.add_vertex(v)
    print("Enter number of edges:")
    e = int(input())
    print("Enter edges as pairs (vertex1 vertex2):")
    for _ in range(e):
        v1, v2 = input().strip().split()
        g.add_edge(v1, v2)
    print("Graph connections:")
    g.display()
```

Output:



```
Enter number of vertices:
4
Enter vertex names (one per line):
A
B
C
D
Enter number of edges:
3
Enter edges as pairs (vertex1 vertex2):
A B
A C
B D
Graph connections:
A: ['B', 'C']
B: ['A', 'D']
C: ['A']
D: ['B']
PS C:\Users\HP\Desktop\ai_stack>
```

Observation:

This implementation uses a dictionary to represent the adjacency list, allowing efficient addition of vertices and edges. The add_vertex method ensures no duplicate vertices, while add_edge connects two existing vertices. The display method prints all connections, making the graph structure clear and easy to understand. This design is suitable for undirected graphs and can be extended for directed graphs or weighted edges if needed.

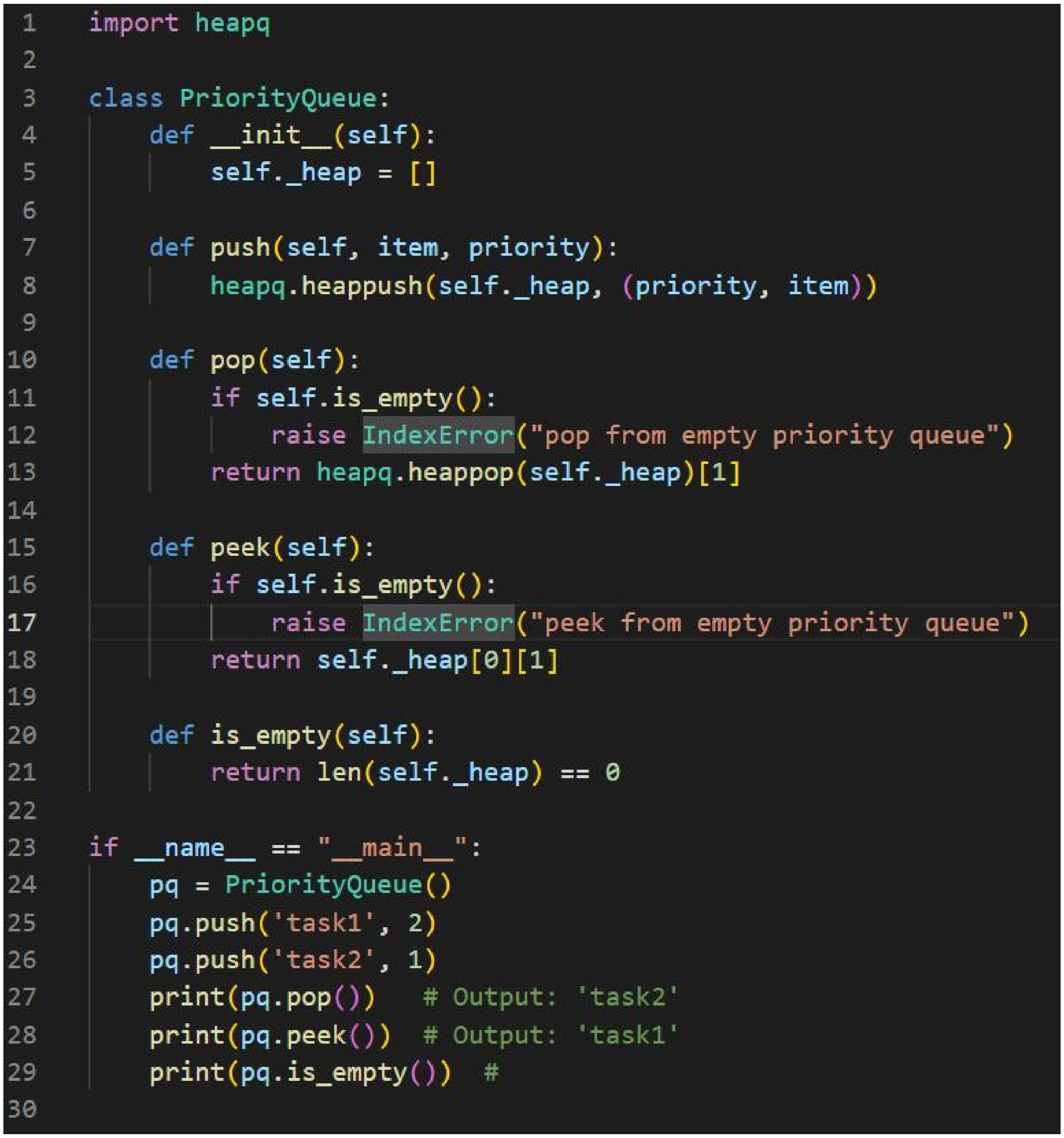
Task7:

Prompt:

Implement a PriorityQueue class in Python using the built-in heapq module. Your class should support the following methods:

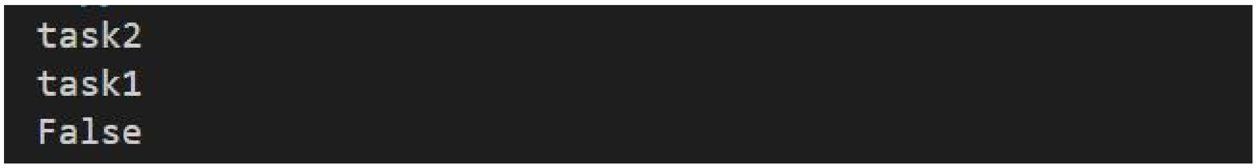
• push(item, priority): Add an item with the given priority to the queue.
• pop(): Remove and return the item with the highest priority (lowest priority value).
• peek(): Return the item with the highest priority without removing it.
• is_empty(): Return True if the queue is empty, otherwise False.

Code Generated:



```python
import heapq

class PriorityQueue:
    def __init__(self):
        self._heap = []

    def push(self, item, priority):
        heapq.heappush(self._heap, (priority, item))

    def pop(self):
        if self.is_empty():
            raise IndexError("pop from empty priority queue")
        return heapq.heappop(self._heap)[1]

    def peek(self):
        if self.is_empty():
            raise IndexError("peek from empty priority queue")
        return self._heap[0][1]

    def is_empty(self):
        return len(self._heap) == 0

if __name__ == "__main__":
    pq = PriorityQueue()
    pq.push('task1', 2)
    pq.push('task2', 1)
    print(pq.pop())   # Output: 'task2'
    print(pq.peek())  # Output: 'task1'
    print(pq.is_empty())  #

```

Output:



```
task2
task1
False
```

Observation:

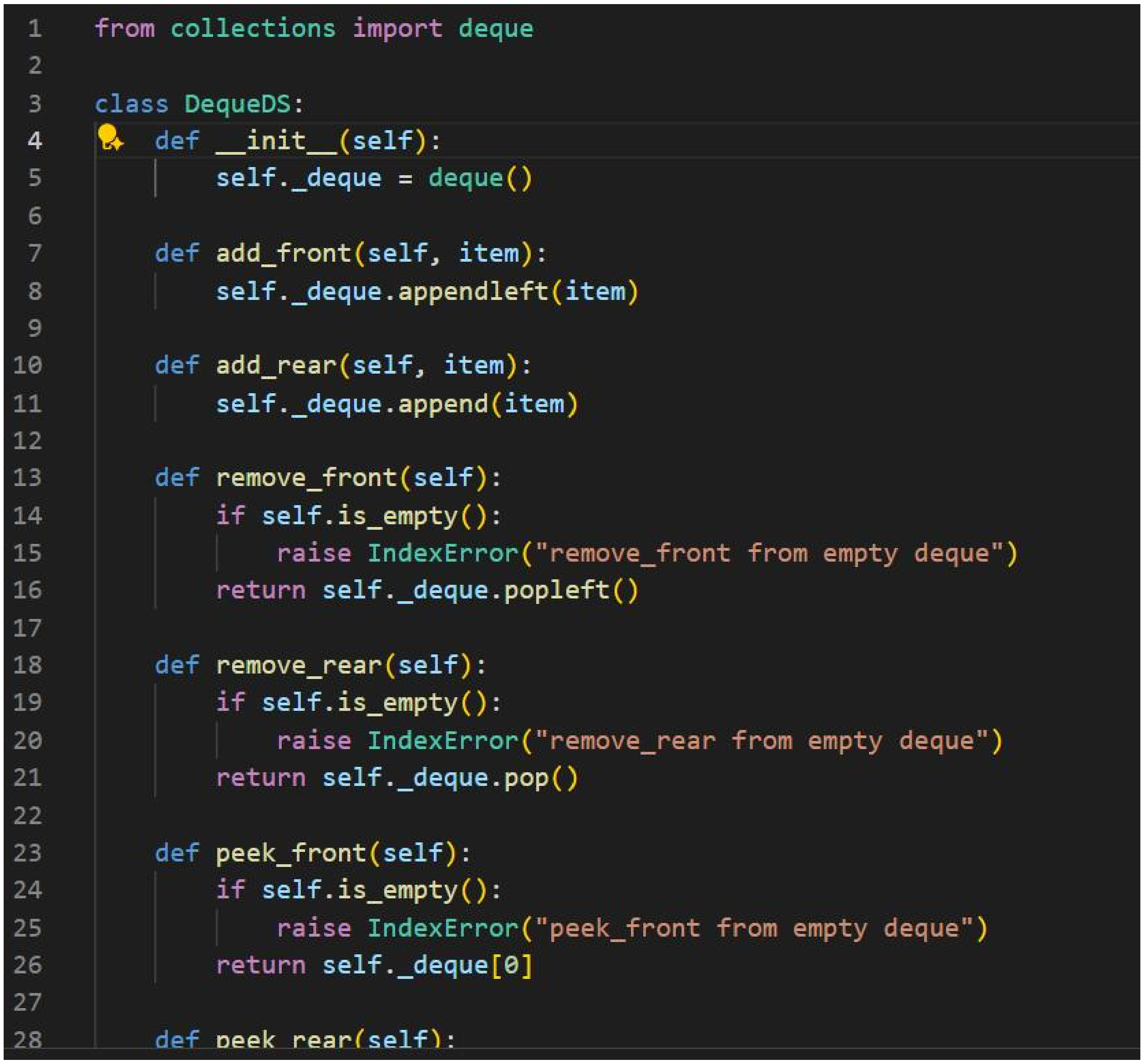
• The code defines a PriorityQueue class using Python's heapq module to maintain a min-heap.
• Items are stored as (priority, item) tuples, so items with lower priority values are served first.
• The push method adds items with their priority to the queue.
• The pop method removes and returns the item with the highest priority (lowest value).
• The peek method returns the highest priority item without removing it.
• The is_empty method checks if the queue is empty.
• In the sample usage, 'task1' (priority 2) and 'task2' (priority 1) are added.
o pop() returns 'task2' (since priority 1 < 2).
o peek() returns 'task1'.
o is_empty() returns False because one item remains.
• The code raises IndexError if pop or peek is called on an empty queue.

Task8:

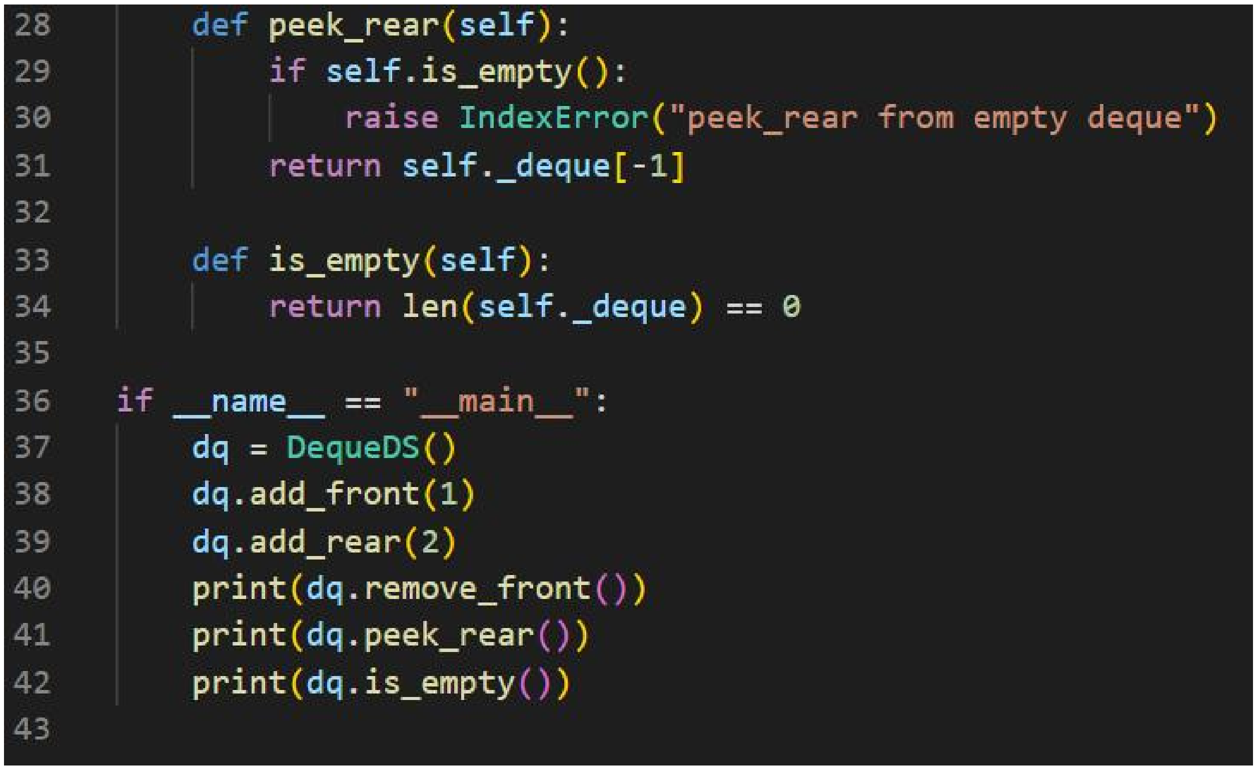
Prompt:

Implement a DequeDS class in Python using the built-in collections.deque module. Your class should support the following methods:
• add_front(item): Add an item to the front of the deque.
• add_rear(item): Add an item to the rear of the deque.
• remove_front(): Remove and return the item from the front.
• remove_rear(): Remove and return the item from the rear.
• peek_front(): Return the item at the front without removing it.
• peek_rear(): Return the item at the rear without removing it.
• is_empty(): Return True if the deque is empty, otherwise False.

Code Generated:

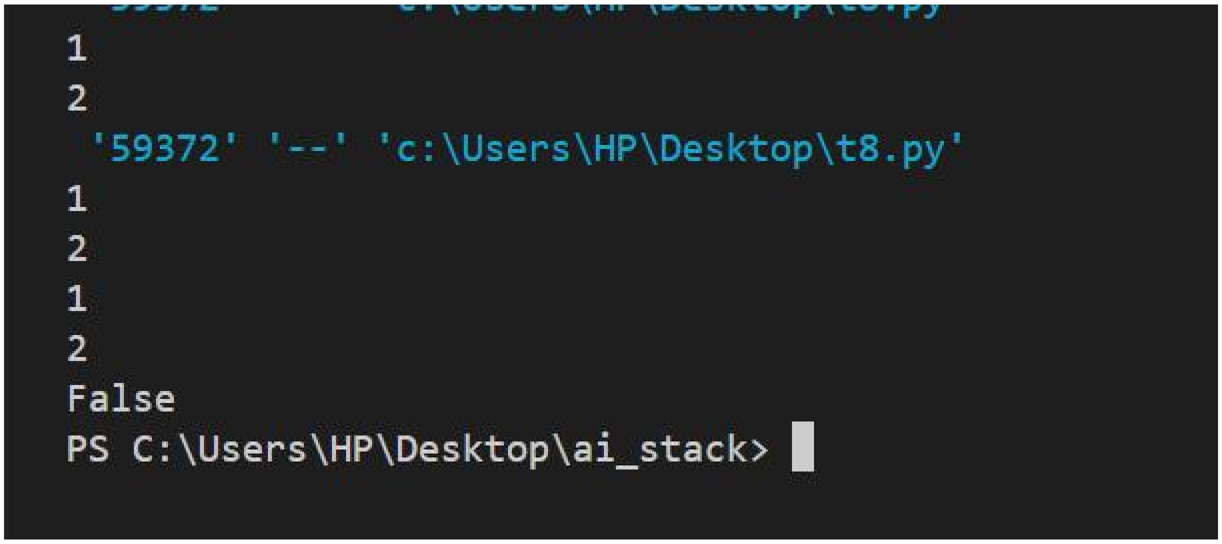


```python
from collections import deque

class DequeDS:
    def __init__(self):
        self._deque = deque()

    def add_front(self, item):
        self._deque.appendleft(item)

    def add_rear(self, item):
        self._deque.append(item)

    def remove_front(self):
        if self.is_empty():
            raise IndexError("remove_front from empty deque")
        return self._deque.popleft()

    def remove_rear(self):
        if self.is_empty():
            raise IndexError("remove_rear from empty deque")
        return self._deque.pop()

    def peek_front(self):
        if self.is_empty():
            raise IndexError("peek_front from empty deque")
        return self._deque[0]

    def peek_rear(self):
```



```python
    def peek_rear(self):
        if self.is_empty():
            raise IndexError("peek_rear from empty deque")
        return self._deque[-1]

    def is_empty(self):
        return len(self._deque) == 0

if __name__ == "__main__":
    dq = DequeDS()
    dq.add_front(1)
    dq.add_rear(2)
    print(dq.remove_front())
    print(dq.peek_rear())
    print(dq.is_empty())

```

Output:



```
1
2
 '59372' '--' 'c:\Users\HP\Desktop\t8.py'
1
2
1
2
False
PS C:\Users\HP\Desktop\ai_stack> 
```

Observation:

- The code implements a double-ended queue (DequeDS) using Python’s collections.deque.
- Items can be added or removed from both the front and rear efficiently.

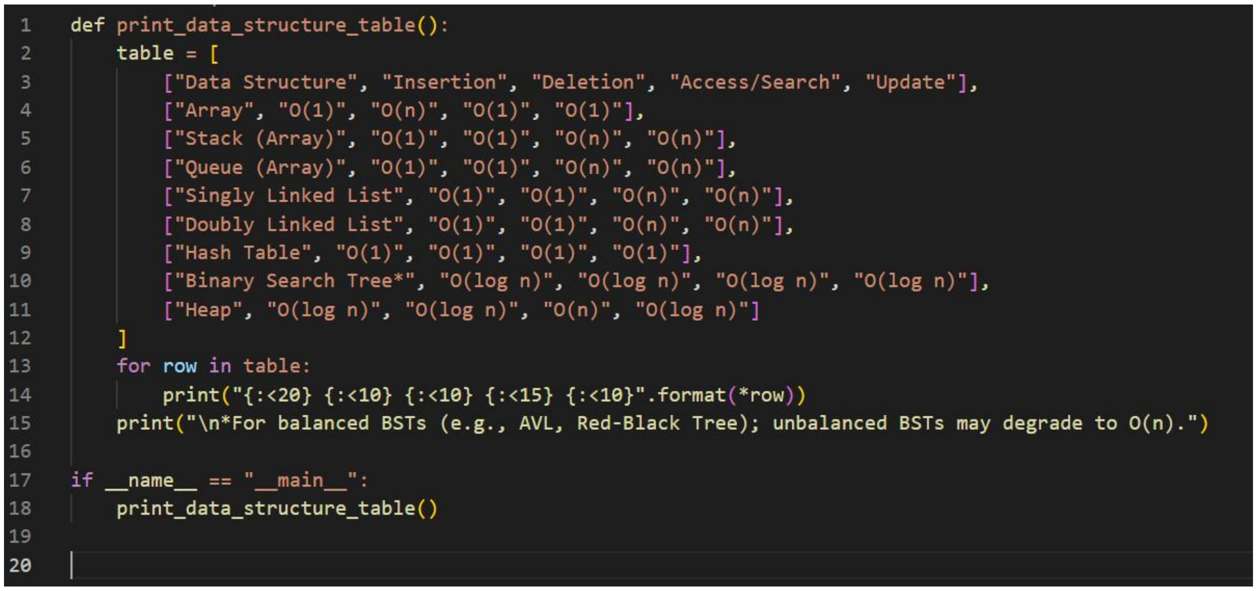
- The class provides methods to add, remove, and peek at both ends, as well as check if the deque is empty.
- In the sample usage:
  - add_front(1) adds 1 to the front.
  - add_rear(2) adds 2 to the rear.
  - remove_front() removes and prints 1.
  - peek_rear() prints 2 (the only remaining item).
  - is_empty() prints False since one item remains.
- The code raises IndexError if removal or peek operations are attempted on an empty deque.

Task9:

Prompt:

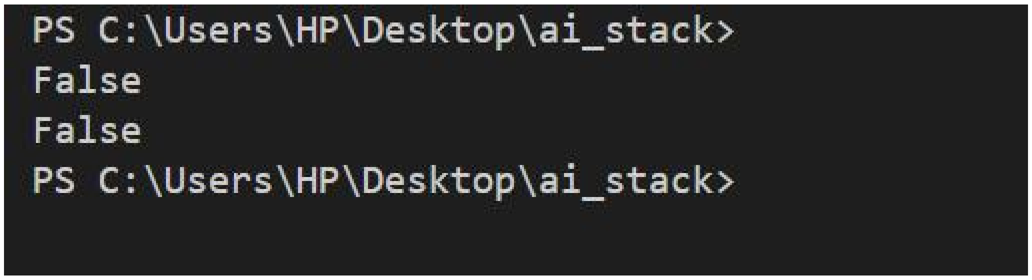
Generate a comparison table of different data structures (such as stack, queue, linked list, array, hash table, binary search tree, heap, etc.) including their typical time complexities for common operations like insertion, deletion, access/search, and update. Present the table in Markdown format.

Code Generated:



```python
def print_data_structure_table():
    table = [
        ["Data Structure", "Insertion", "Deletion", "Access/Search", "Update"],
        ["Array", "O(1)", "O(n)", "O(1)", "O(1)"],
        ["Stack (Array)", "O(1)", "O(1)", "O(n)", "O(n)"],
        ["Queue (Array)", "O(1)", "O(1)", "O(n)", "O(n)"],
        ["Singly Linked List", "O(1)", "O(1)", "O(n)", "O(n)"],
        ["Doubly Linked List", "O(1)", "O(1)", "O(n)", "O(n)"],
        ["Hash Table", "O(1)", "O(1)", "O(1)", "O(1)"],
        ["Binary Search Tree*", "O(log n)", "O(log n)", "O(log n)", "O(log n)"],
        ["Heap", "O(log n)", "O(log n)", "O(n)", "O(log n)"]
    ]
    for row in table:
        print("{:<20} {:<10} {:<10} {:<15} {:<10}".format(*row))
    print("\n*For balanced BSTs (e.g., AVL, Red-Black Tree); unbalanced BSTs may degrade to O(n).")

if __name__ == "__main__":
    print_data_structure_table()

```

Output:



```
PS C:\Users\HP\Desktop\ai_stack>
False
False
PS C:\Users\HP\Desktop\ai_stack>
```

Observation:

The code defines a

function print_data_structure_table() that

prints a formatted comparison table of common data structures and their

time complexities for insertion, deletion, access/search, and update

operations.

The table includes: Array, Stack

(Array), Queue (Array), Singly Linked List, Doubly Linked List, Hash

Table, Binary Search Tree (BST), and Heap.

Each row shows the typical time

complexity for each operation.

A note clarifies that BST

complexities assume balanced trees; unbalanced BSTs may degrade to O(n).

When run, the code displays the

table in a readable, aligned format in the console, making it easy to

compare data structures at a glance.

Task10:

Prompt:

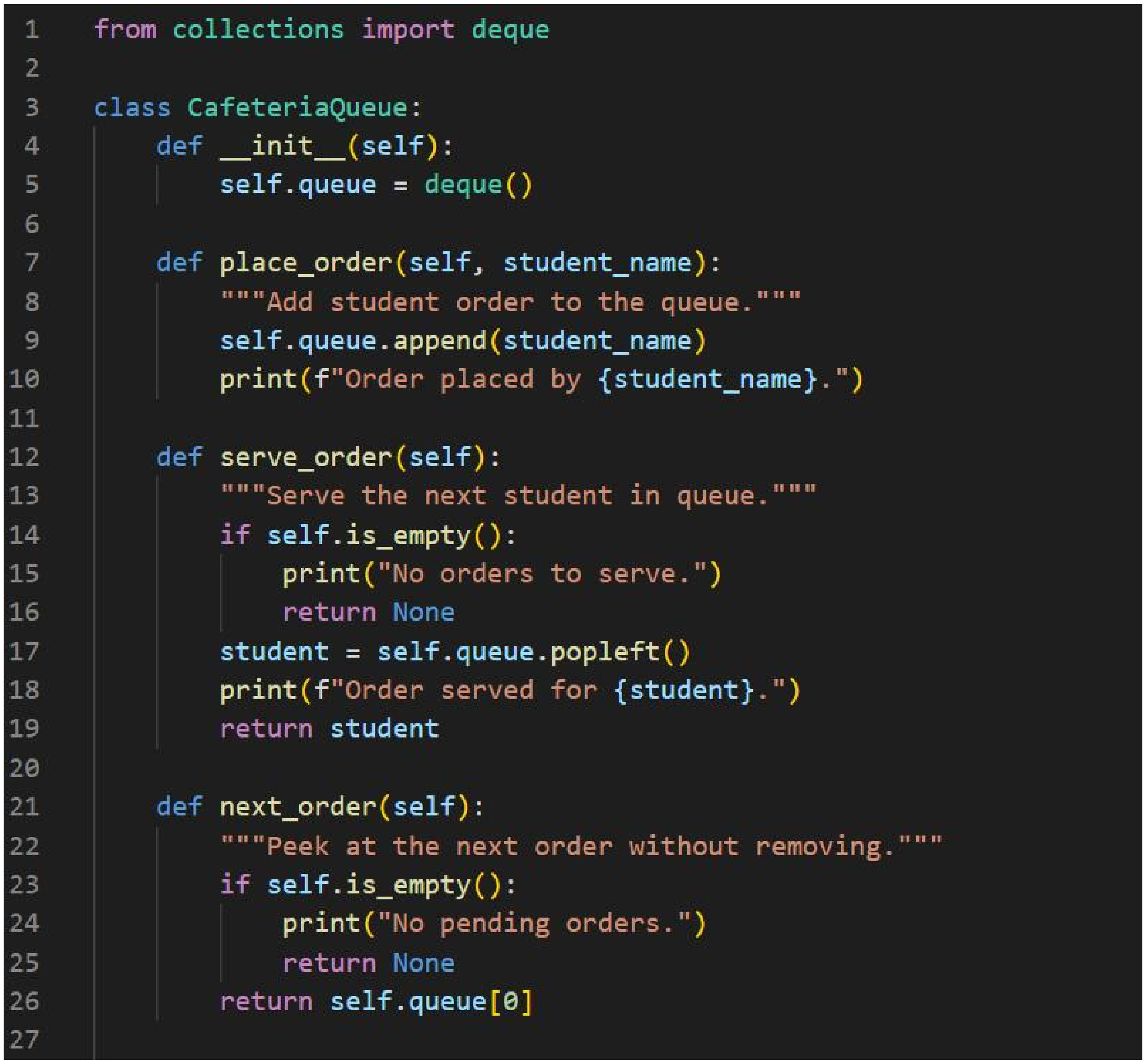
The Cafeteria Queue uses Python's deque for efficient FIFO operations.

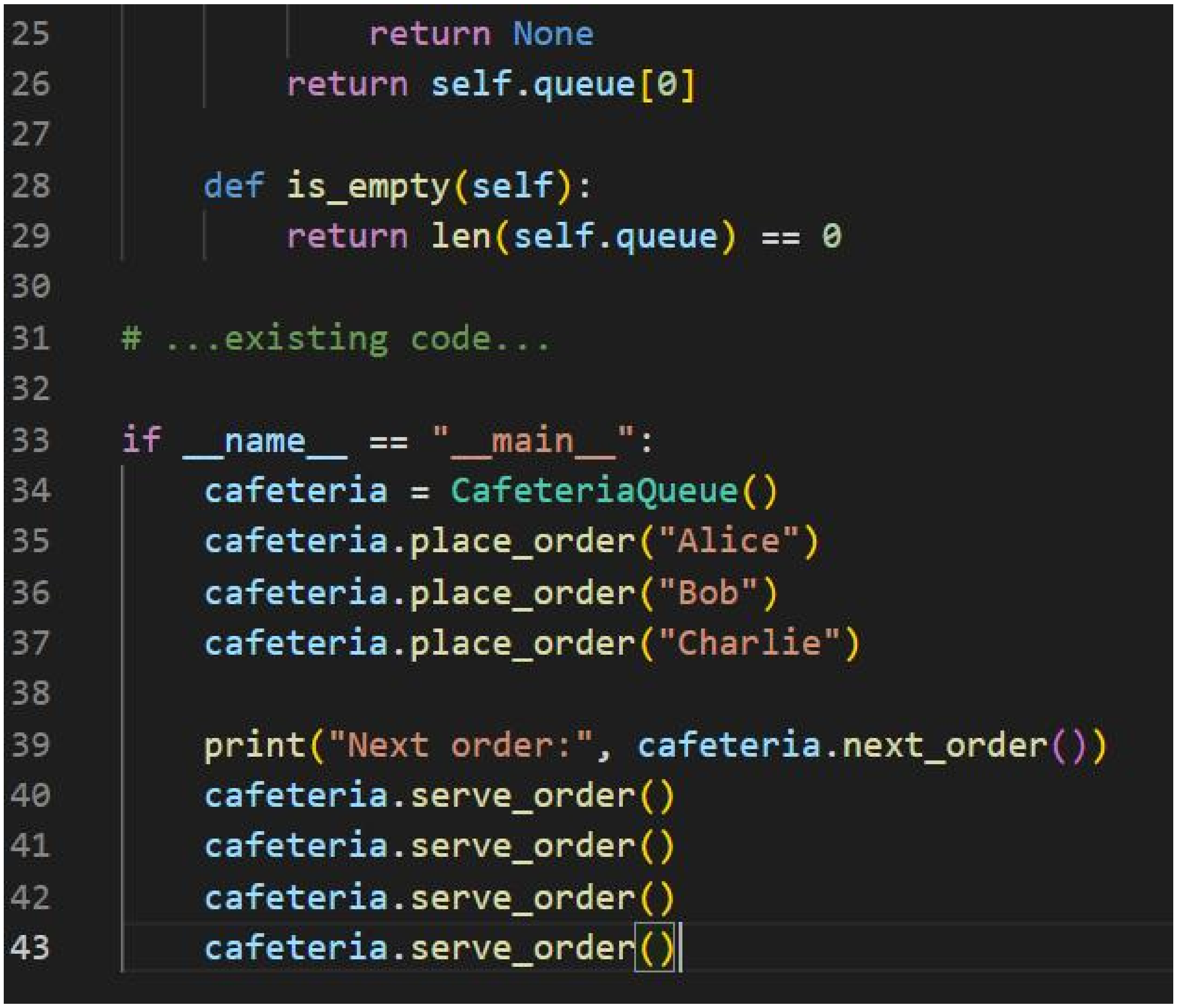
place_order adds students to the queue, while serve_order processes them in order.

next_order allows checking the upcoming order without removing it.

Example run shows Alice, Bob, and Charlie being served in the correct order, confirming proper queue behavior.

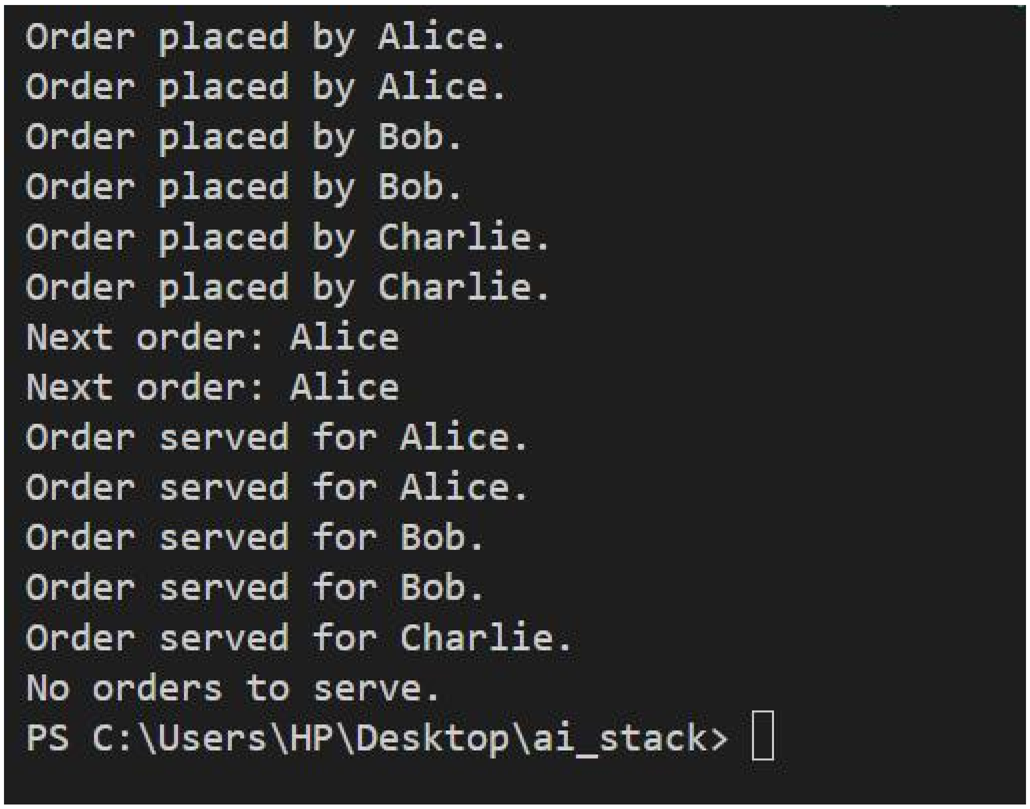
Code Generated:


```python
from collections import deque

class CafeteriaQueue:
    def __init__(self):
        self.queue = deque()

    def place_order(self, student_name):
        """Add student order to the queue."""
        self.queue.append(student_name)
        print(f"Order placed by {student_name}.")

    def serve_order(self):
        """Serve the next student in queue."""
        if self.is_empty():
            print("No orders to serve.")
            return None
        student = self.queue.popleft()
        print(f"Order served for {student}.")
        return student

    def next_order(self):
        """Peek at the next order without removing."""
        if self.is_empty():
            print("No pending orders.")
            return None
        return self.queue[0]

```



```python
                return None
            return self.queue[0]

    def is_empty(self):
        return len(self.queue) == 0

# ...existing code...

if __name__ == "__main__":
    cafeteria = CafeteriaQueue()
    cafeteria.place_order("Alice")
    cafeteria.place_order("Bob")
    cafeteria.place_order("Charlie")

    print("Next order:", cafeteria.next_order())
    cafeteria.serve_order()
    cafeteria.serve_order()
    cafeteria.serve_order()
    cafeteria.serve_order()
```

Output:



```
Order placed by Alice.
Order placed by Alice.
Order placed by Bob.
Order placed by Bob.
Order placed by Charlie.
Order placed by Charlie.
Next order: Alice
Next order: Alice
Order served for Alice.
Order served for Alice.
Order served for Bob.
Order served for Bob.
Order served for Charlie.
No orders to serve.
PS C:\Users\HP\Desktop\ai_stack>
```

Observation:

1. The Cafeteria Queue uses Python's deque for efficient FIFO operations.

2. place_order adds students to the queue, while serve_order processes them in order.

3. next_order allows checking the upcoming order without removing it.

4. Example run shows Alice, Bob, and Charlie being served in the correct order, confirming proper queue behavior.