AI ASSISSTED LAB 11.1

NAME:B.VISHNU VARDHAN

BATCH NO:06

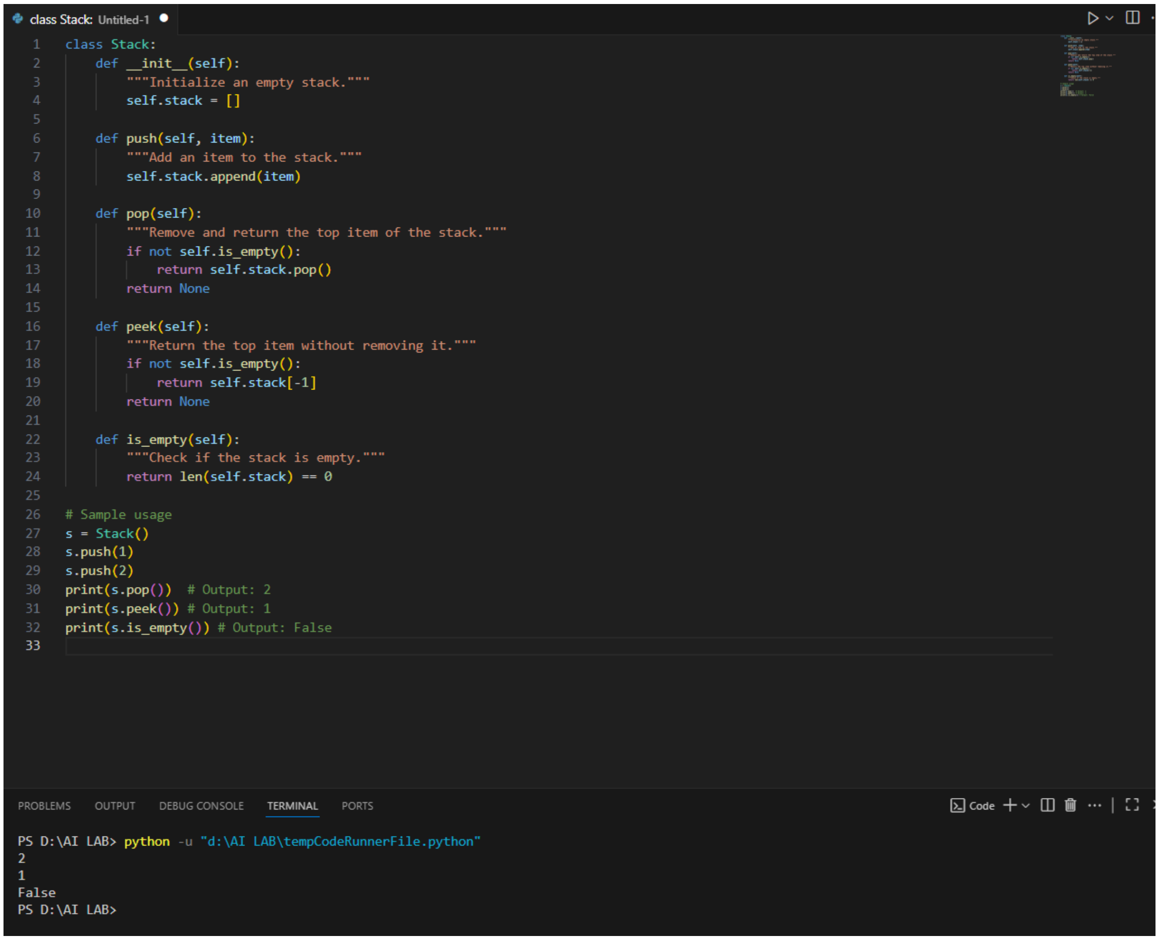
ROLL NO:2403A510F2

STACK IMPLEMENTATION

PROMPT

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

CODE



```python
class Stack:
    def __init__(self):
        """Initialize an empty stack."""
        self.stack = []

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

    def pop(self):
        """Remove and return the top item of the stack."""
        if not self.is_empty():
            return self.stack.pop()
        return None

    def peek(self):
        """Return the top item without removing it."""
        if not self.is_empty():
            return self.stack[-1]
        return None

    def is_empty(self):
        """Check if the stack is empty."""
        return len(self.stack) == 0

# Sample usage
s = Stack()
s.push(1)
s.push(2)
print(s.pop())  # Output: 2
print(s.peek()) # Output: 1
print(s.is_empty()) # Output: False
```

```
PS D:\AI LAB> python -u "d:\AI LAB\tempCodeRunnerFile.python"
2
1
False
PS D:\AI LAB>
```

OUTPUT

2
1
False

OBSERVATION

AI ASSISSTED LAB 11.1

Correct LIFO behavior demonstrated.

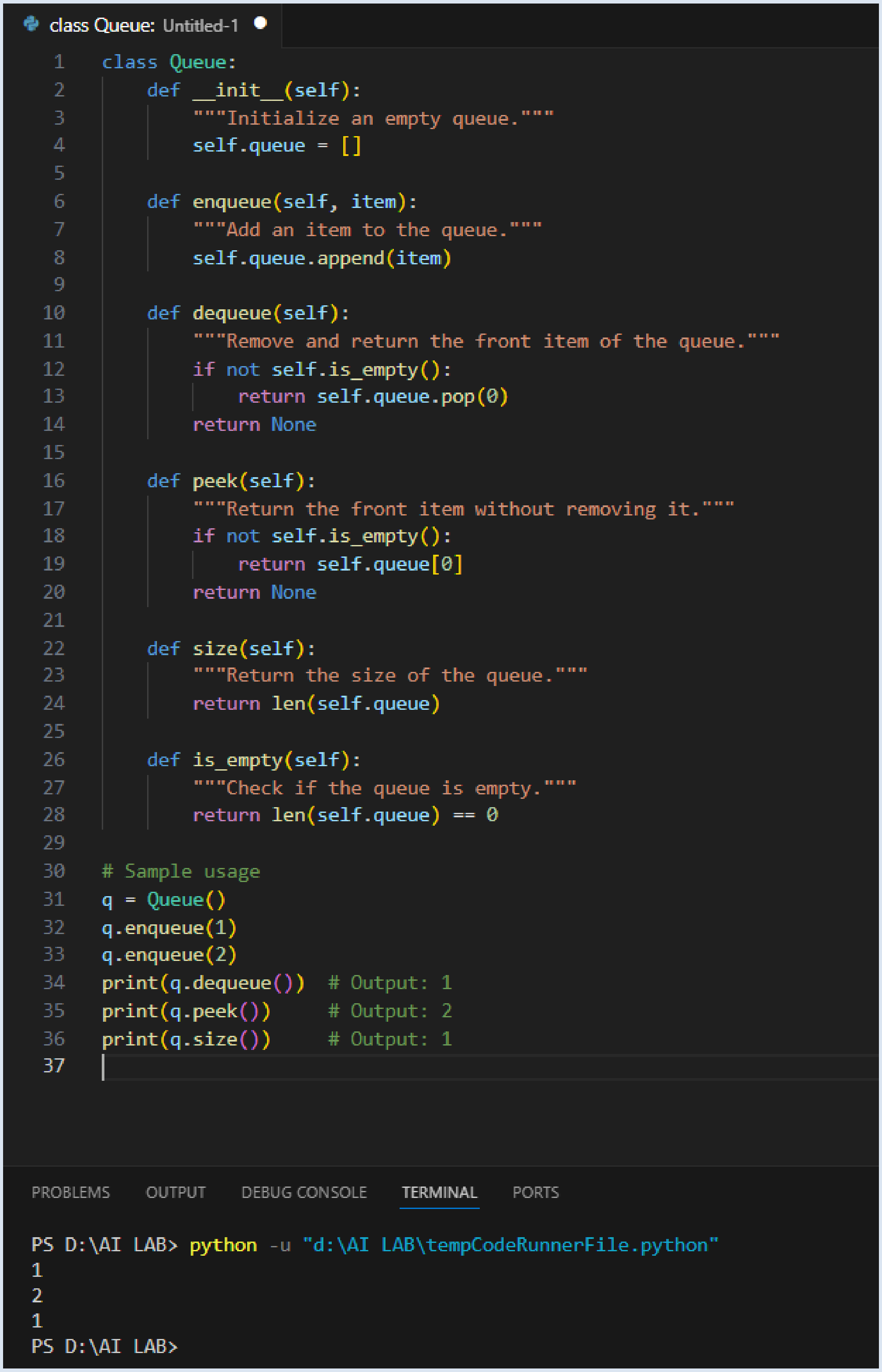
QUEUE IMPLEMENTATION

PROMPT

Use AI to implement a Queue using Python lists.

CODE

AI ASSISSTED LAB 11.1



```python
class Queue:
    def __init__(self):
        """Initialize an empty queue."""
        self.queue = []

    def enqueue(self, item):
        """Add an item to the queue."""
        self.queue.append(item)

    def dequeue(self):
        """Remove and return the front item of the queue."""
        if not self.is_empty():
            return self.queue.pop(0)
        return None

    def peek(self):
        """Return the front item without removing it."""
        if not self.is_empty():
            return self.queue[0]
        return None

    def size(self):
        """Return the size of the queue."""
        return len(self.queue)

    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.queue) == 0

# Sample usage
q = Queue()
q.enqueue(1)
q.enqueue(2)
print(q.dequeue())  # Output: 1
print(q.peek())     # Output: 2
print(q.size())     # Output: 1
```

```
PS D:\AI LAB> python -u "d:\AI LAB\tempCodeRunnerFile.python"
1
2
1
PS D:\AI LAB>
```

OUTPUT

1
2
1

AI ASSISSTED LAB 11.1

OBSERVATION

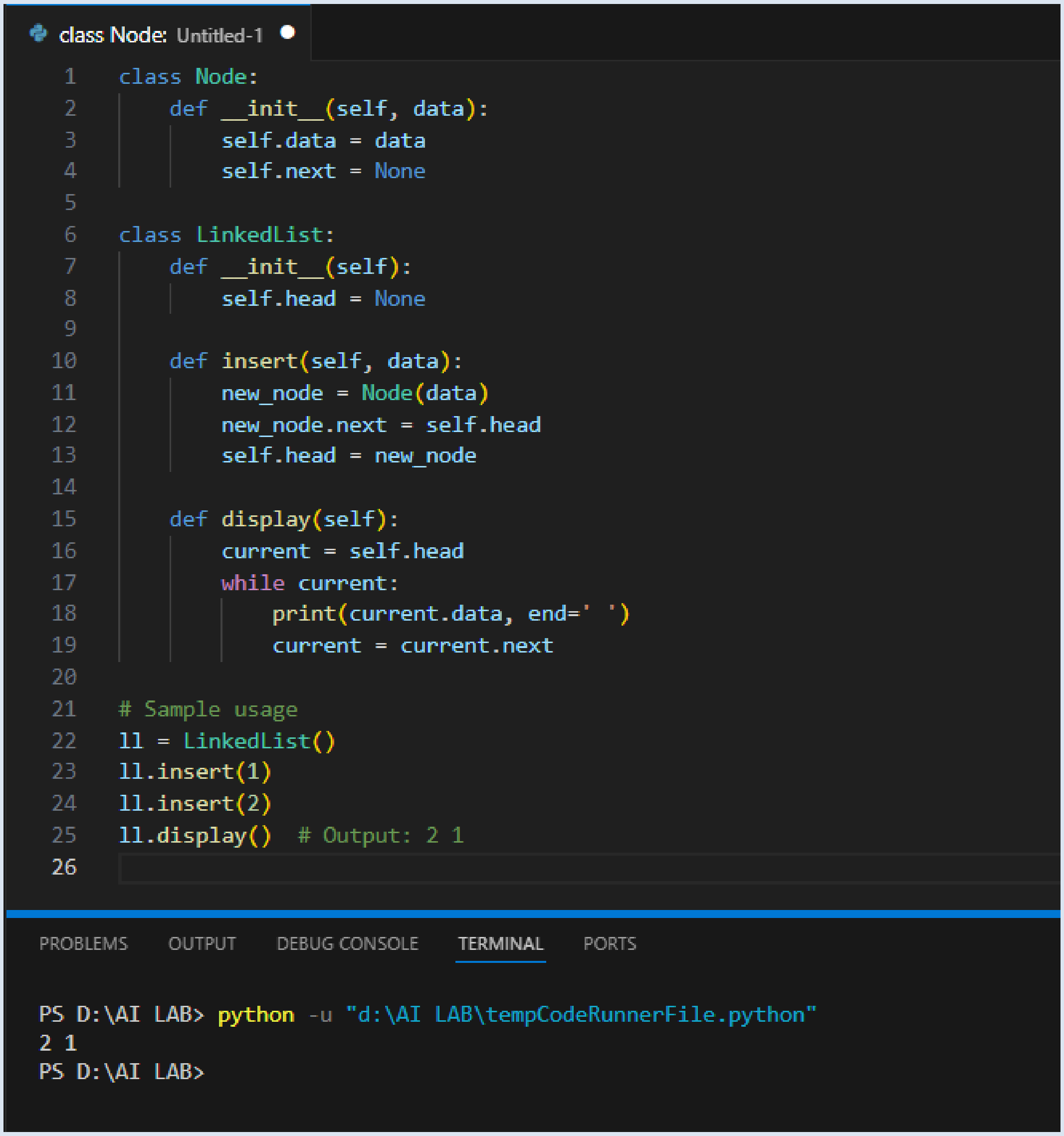
FIFO behavior works as expected.

LINKED LIST IMPLEMENTATION

PROMPT

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

CODE



```python
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)
        new_node.next = self.head
        self.head = new_node

    def display(self):
        current = self.head
        while current:
            print(current.data, end=' ')
            current = current.next

# Sample usage
ll = LinkedList()
ll.insert(1)
ll.insert(2)
ll.display()  # Output: 2 1
```

```
PS D:\AI LAB> python -u "d:\AI LAB\tempCodeRunnerFile.python"
2 1
PS D:\AI LAB>
```

OUTPUT

2 1

OBSERVATION

AI ASSISSTED LAB 11.1

Linked list inserts and displays correctly.

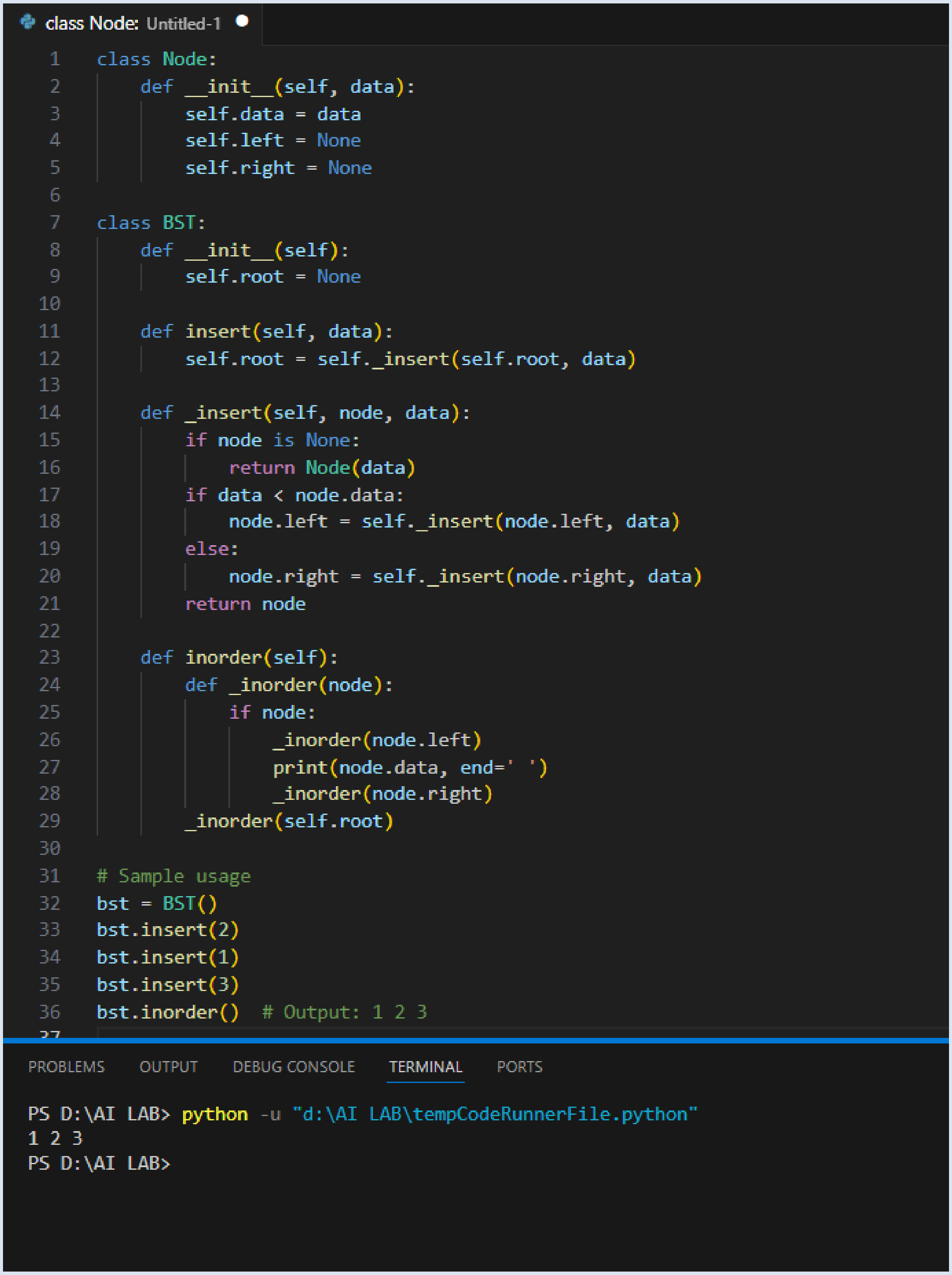
BINARY SEARCH TREE (BST) IMPLEMENTATION

PROMPT

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

CODE

AI ASSISSTED LAB 11.1



```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, data):
        self.root = self._insert(self.root, data)

    def _insert(self, node, data):
        if node is None:
            return Node(data)
        if data < node.data:
            node.left = self._insert(node.left, data)
        else:
            node.right = self._insert(node.right, data)
        return node

    def inorder(self):
        def _inorder(node):
            if node:
                _inorder(node.left)
                print(node.data, end=' ')
                _inorder(node.right)
        _inorder(self.root)

# Sample usage
bst = BST()
bst.insert(2)
bst.insert(1)
bst.insert(3)
bst.inorder()  # Output: 1 2 3
```

```
PS D:\AI LAB> python -u "d:\AI LAB\tempCodeRunnerFile.python"
1 2 3
PS D:\AI LAB>
```

OUTPUT

1 2 3

OBSERVATION

BST correctly inserts and traverses in-order.

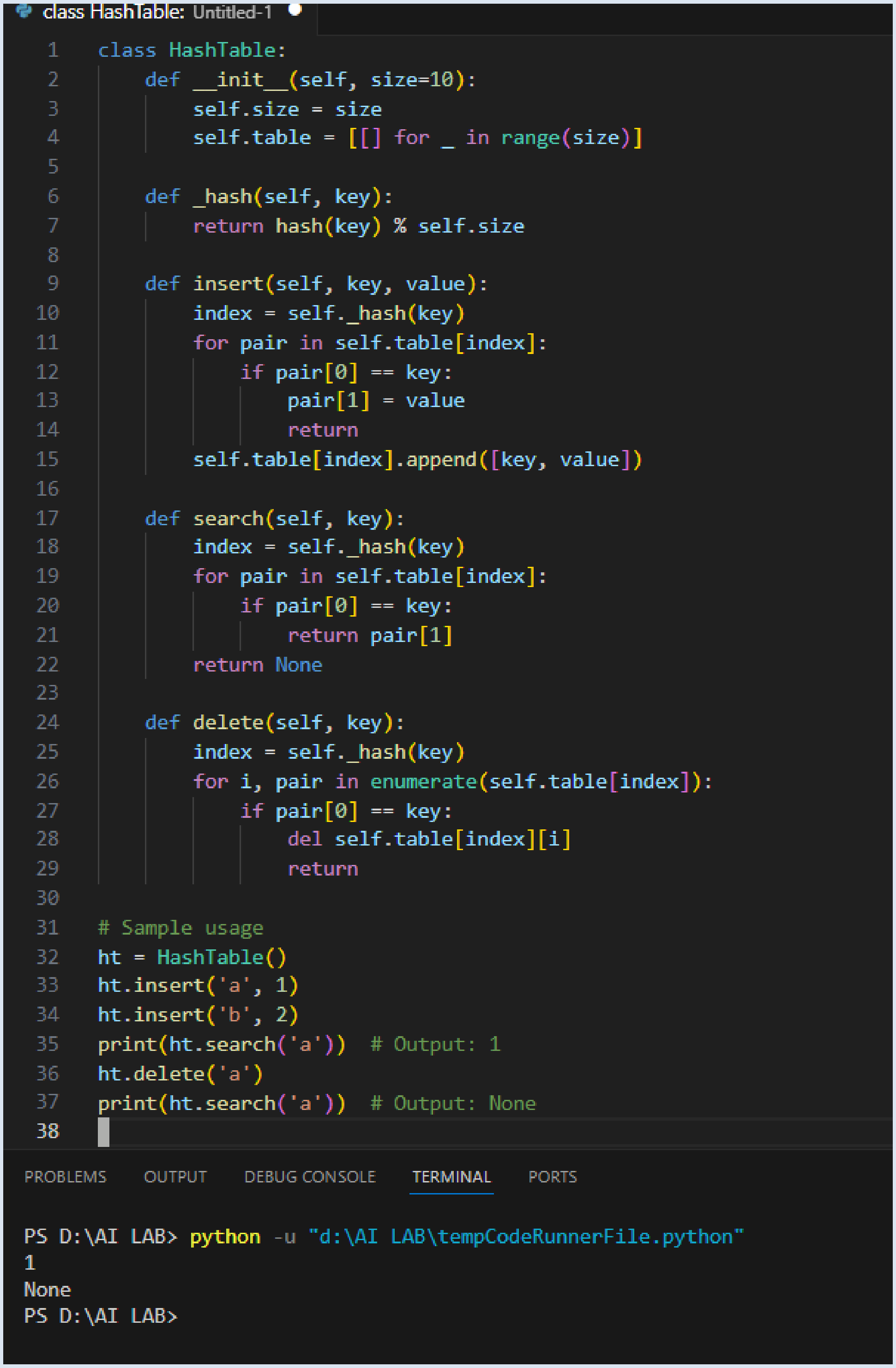
AI ASSISSTED LAB 11.1

HASH TABLE IMPLEMENTATION

PROMPT

Use AI to implement a hash table with insert, search, and delete using chaining for collision handling.

CODE



```python
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 pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return
        self.table[index].append([key, value])

    def search(self, key):
        index = self._hash(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return None

    def delete(self, key):
        index = self._hash(key)
        for i, pair in enumerate(self.table[index]):
            if pair[0] == key:
                del self.table[index][i]
                return

# Sample usage
ht = HashTable()
ht.insert('a', 1)
ht.insert('b', 2)
print(ht.search('a'))  # Output: 1
ht.delete('a')
print(ht.search('a'))  # Output: None

```

```
PS D:\AI LAB> python -u "d:\AI LAB\tempCodeRunnerFile.python"
1
None
PS D:\AI LAB>
```

AI ASSISSTED LAB 11.1

OUTPUT

1
None

OBSERVATION

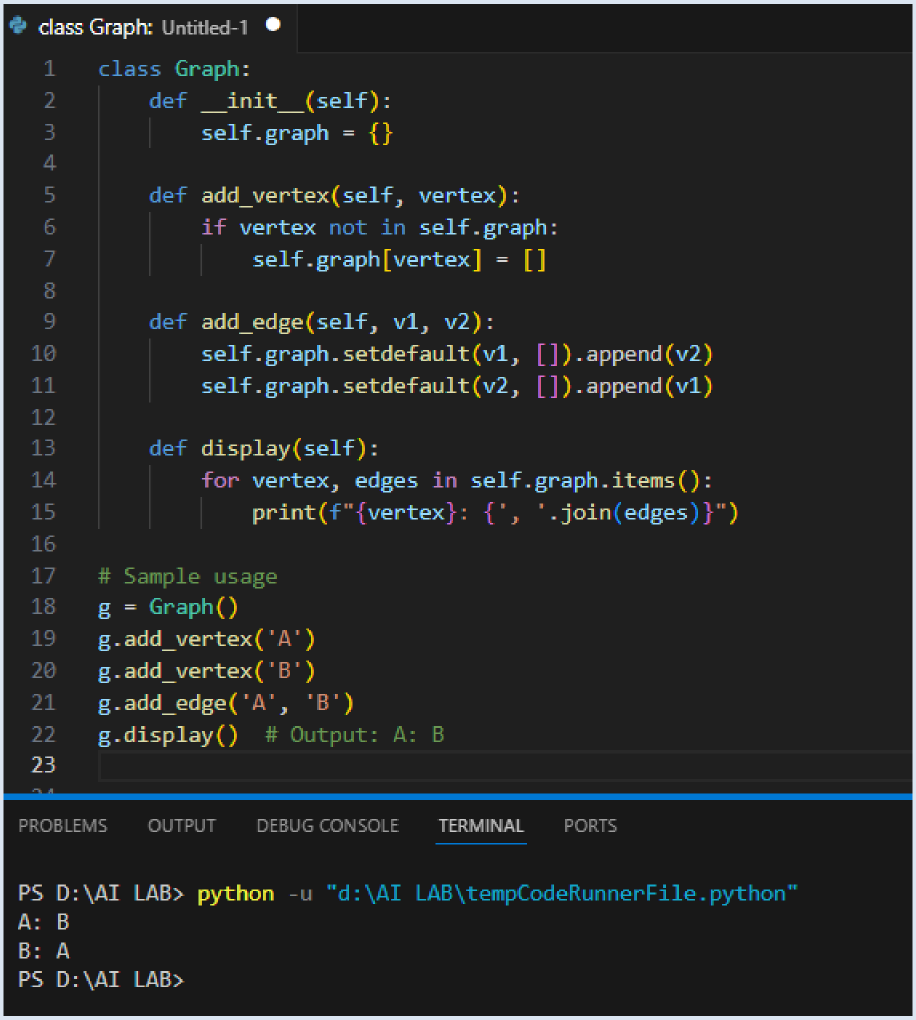
Hash table inserts, searches, and deletes with chaining as expected.

GRAPH REPRESENTATION IMPLEMENTATION

PROMPT

Use AI to implement a graph using an adjacency list.

CODE



```python
class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, v1, v2):
        self.graph.setdefault(v1, []).append(v2)
        self.graph.setdefault(v2, []).append(v1)

    def display(self):
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {', '.join(edges)}")

# Sample usage
g = Graph()
g.add_vertex('A')
g.add_vertex('B')
g.add_edge('A', 'B')
g.display()  # Output: A: B

```

```
PS D:\AI LAB> python -u "d:\AI LAB\tempCodeRunnerFile.python"
A: B
B: A
PS D:\AI LAB>
```

OUTPUT

A: B
B: A

AI ASSISSTED LAB 11.1

OBSERVATION

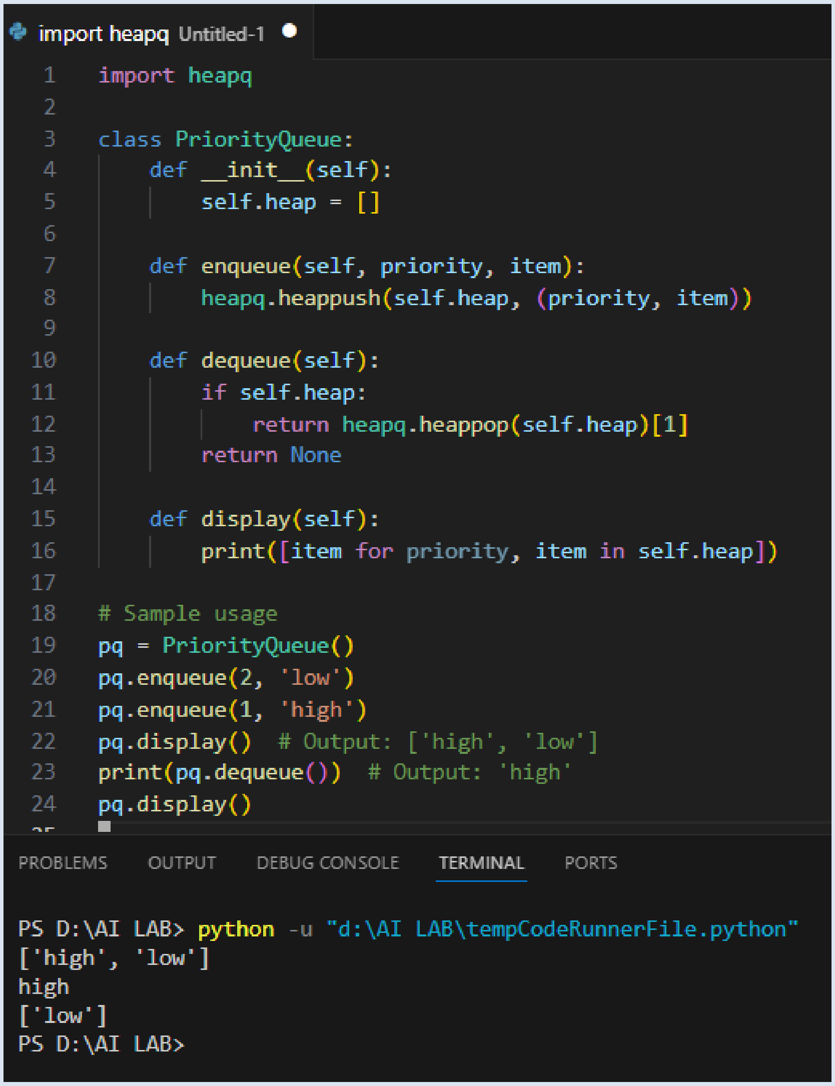
Graph is correctly represented using adjacency list.

PRIORITY QUEUE IMPLEMENTATION

PROMPT

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

CODE



```python
import heapq

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

    def enqueue(self, priority, item):
        heapq.heappush(self.heap, (priority, item))

    def dequeue(self):
        if self.heap:
            return heapq.heappop(self.heap)[1]
        return None

    def display(self):
        print([item for priority, item in self.heap])

# Sample usage
pq = PriorityQueue()
pq.enqueue(2, 'low')
pq.enqueue(1, 'high')
pq.display()  # Output: ['high', 'low']
print(pq.dequeue())  # Output: 'high'
pq.display()
```

```
PS D:\AI LAB> python -u "d:\AI LAB\tempCodeRunnerFile.python"
['high', 'low']
high
['low']
PS D:\AI LAB>
```

OUTPUT

['high', 'low']
high
['low']

OBSERVATION

AI ASSISSTED LAB 11.1

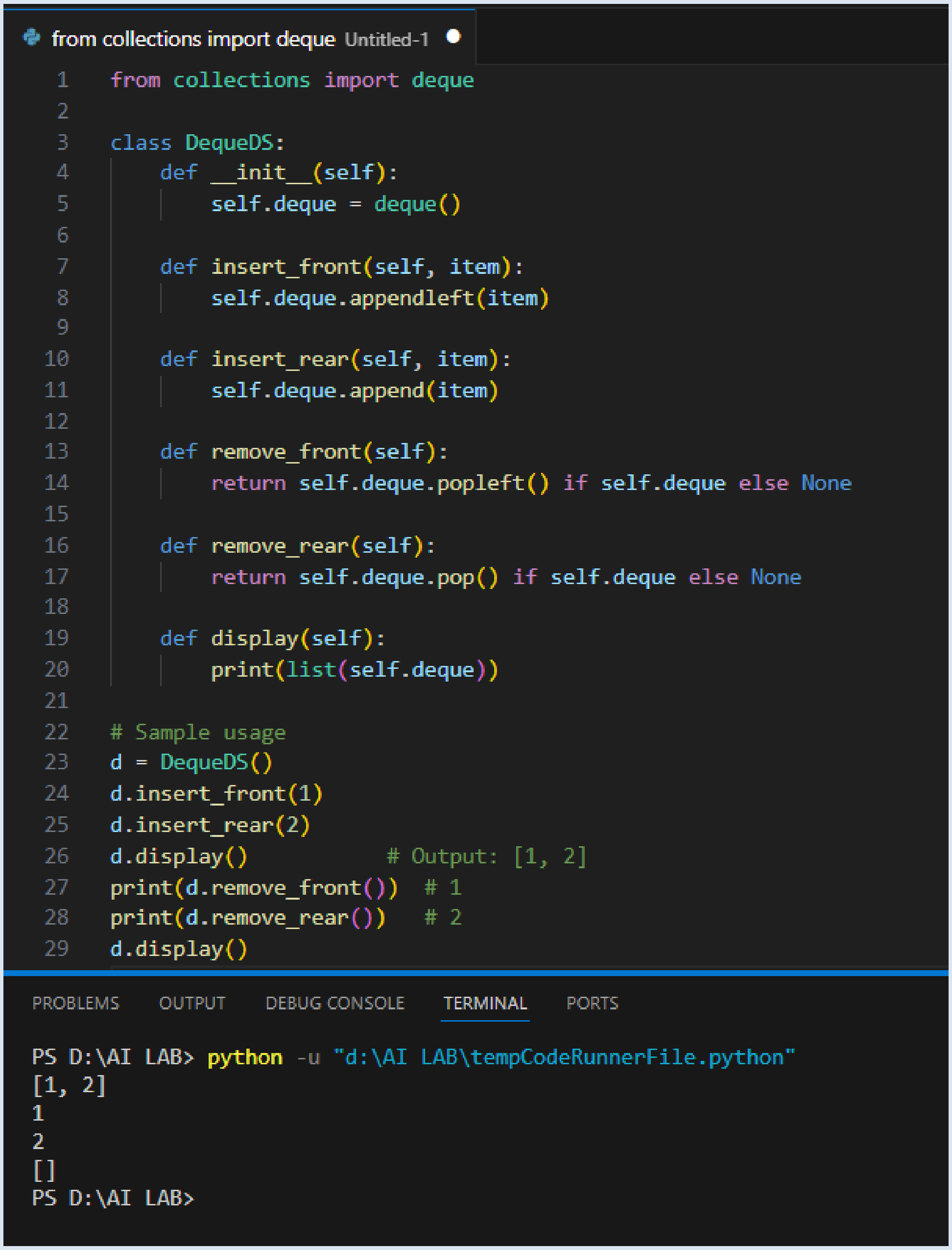
Priority queue orders items by priority correctly.

DEQUE IMPLEMENTATION

PROMPT

Use AI to implement a double-ended queue using collections.deque.

CODE



```python
from collections import deque

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

    def insert_front(self, item):
        self.deque.appendleft(item)

    def insert_rear(self, item):
        self.deque.append(item)

    def remove_front(self):
        return self.deque.popleft() if self.deque else None

    def remove_rear(self):
        return self.deque.pop() if self.deque else None

    def display(self):
        print(list(self.deque))

# Sample usage
d = DequeDS()
d.insert_front(1)
d.insert_rear(2)
d.display()           # Output: [1, 2]
print(d.remove_front())  # 1
print(d.remove_rear())   # 2
d.display()
```

```
PS D:\AI LAB> python -u "d:\AI LAB\tempCodeRunnerFile.python"
[1, 2]
1
2
[]
PS D:\AI LAB>
```

OUTPUT

```
[1, 2]
1
```

AI ASSISSTED LAB 11.1

2
[]

OBSERVATION

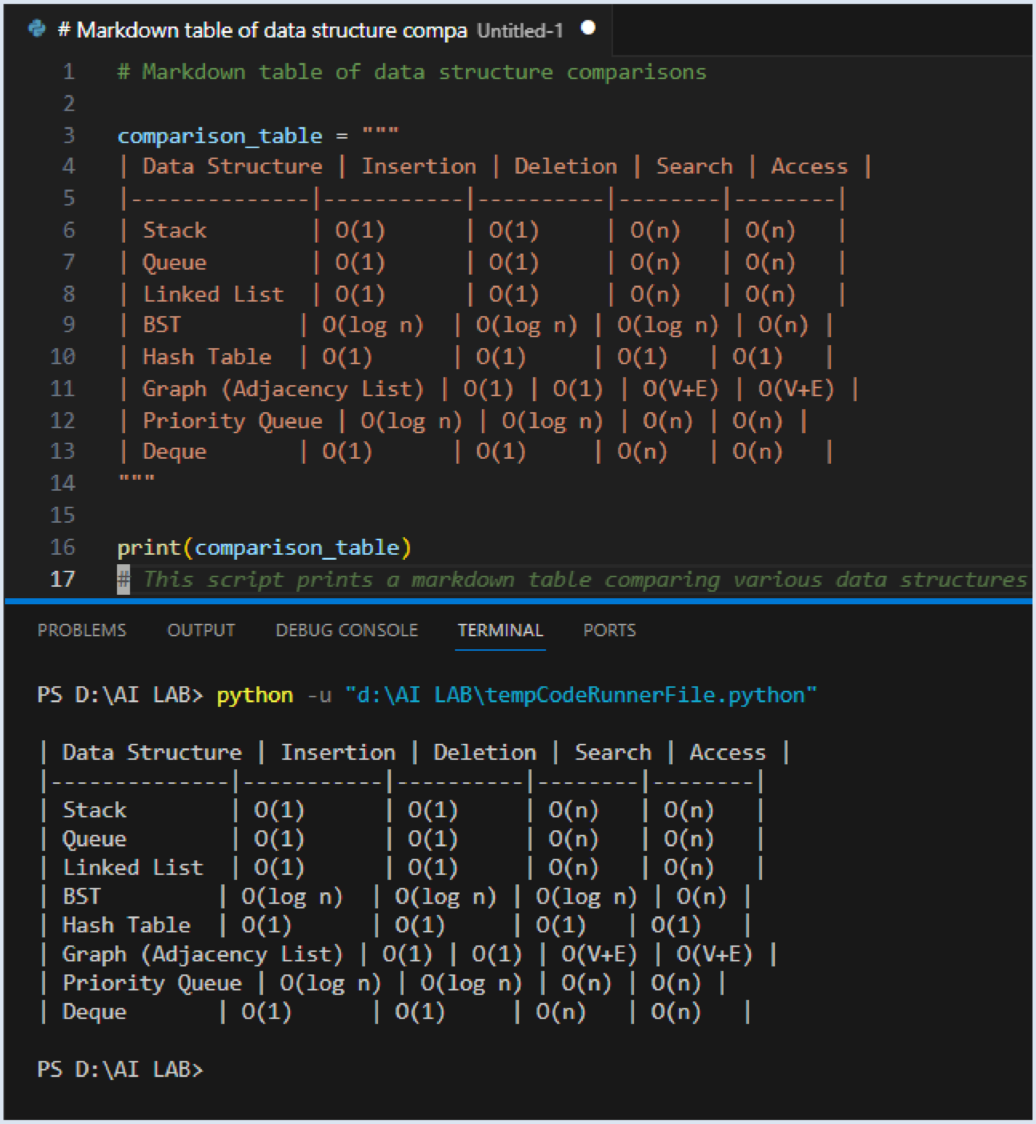
Deque supports double-ended operations correctly.

DATA STRUCTURE COMPARISON

PROMPT

Use AI to generate a comparison table of different data structures including time complexities.

CODE



```
# Markdown table of data structure comparisons

comparison_table = """
| Data Structure | Insertion | Deletion | Search | Access |
|--------------|-----------|----------|--------|--------|
| Stack        | O(1)      | O(1)     | O(n)   | O(n)   |
| Queue        | O(1)      | O(1)     | O(n)   | O(n)   |
| Linked List  | O(1)      | O(1)     | O(n)   | O(n)   |
| BST        | O(log n)  | O(log n) | O(log n) | O(n) |
| Hash Table | O(1)      | O(1)     | O(1)   | O(1)   |
| Graph (Adjacency List) | O(1) | O(1) | O(V+E) | O(V+E) |
| Priority Queue | O(log n) | O(log n) | O(n) | O(n) |
| Deque      | O(1)      | O(1)     | O(n)   | O(n)   |
"""

print(comparison_table)
# This script prints a markdown table comparing various data structures
```

```
PS D:\AI LAB> python -u "d:\AI LAB\tempCodeRunnerFile.python"

| Data Structure | Insertion | Deletion | Search | Access |
|--------------|-----------|----------|--------|--------|
| Stack        | O(1)      | O(1)     | O(n)   | O(n)   |
| Queue        | O(1)      | O(1)     | O(n)   | O(n)   |
| Linked List  | O(1)      | O(1)     | O(n)   | O(n)   |
| BST        | O(log n)  | O(log n) | O(log n) | O(n) |
| Hash Table | O(1)      | O(1)     | O(1)   | O(1)   |
| Graph (Adjacency List) | O(1) | O(1) | O(V+E) | O(V+E) |
| Priority Queue | O(log n) | O(log n) | O(n) | O(n) |
| Deque      | O(1)      | O(1)     | O(n)   | O(n)   |

PS D:\AI LAB>
```

OUTPUT

AI ASSISSTED LAB 11.1

Data Structure	Insertion	Deletion	Search	Access
Stack	O(1)	O(1)	O(n)	O(n)
Queue	O(1)	O(1)	O(n)	O(n)
Linked List	O(1)	O(1)	O(n)	O(n)
BST	O(log n)	O(log n)	O(log n)	O(n)
Hash Table	O(1)	O(1)	O(1)	O(1)
Graph (Adjacency List)	O(1)	O(1)	O(V+E)	O(V+E)
Priority Queue	O(log n)	O(log n)	O(n)	O(n)
Deque	O(1)	O(1)	O(n)	O(n)

OBSERVATION

Clear comparison of time complexities provided.

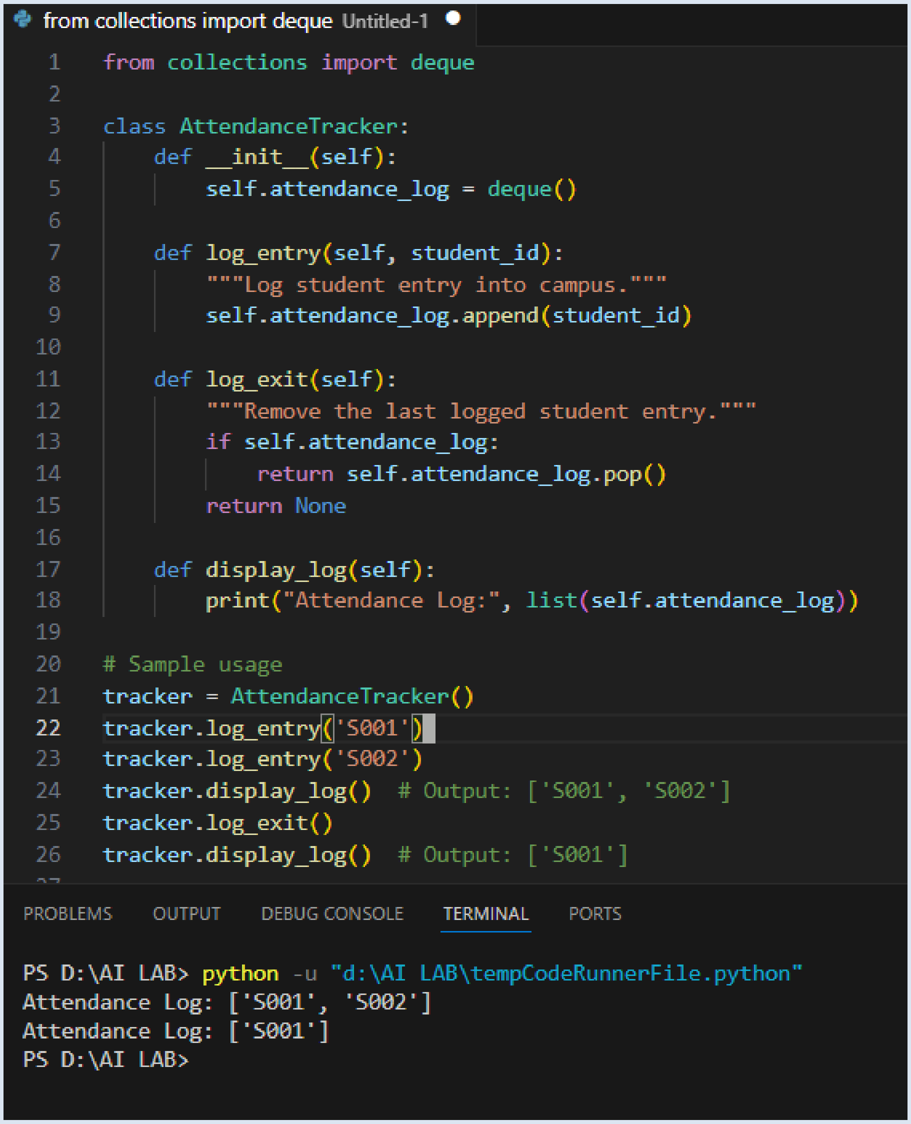
REAL-TIME APPLICATION CHALLENGE

PROMPT

Implement Student Attendance Tracking using an appropriate data structure.

CODE

AI ASSISSTED LAB 11.1



```python
from collections import deque

class AttendanceTracker:
    def __init__(self):
        self.attendance_log = deque()

    def log_entry(self, student_id):
        """Log student entry into campus."""
        self.attendance_log.append(student_id)

    def log_exit(self):
        """Remove the last logged student entry."""
        if self.attendance_log:
            return self.attendance_log.pop()
        return None

    def display_log(self):
        print("Attendance Log:", list(self.attendance_log))

# Sample usage
tracker = AttendanceTracker()
tracker.log_entry('S001')
tracker.log_entry('S002')
tracker.display_log()  # Output: ['S001', 'S002']
tracker.log_exit()
tracker.display_log()  # Output: ['S001']
```

```
PS D:\AI LAB> python -u "d:\AI LAB\tempCodeRunnerFile.python"
Attendance Log: ['S001', 'S002']
Attendance Log: ['S001']
PS D:\AI LAB>
```

OUTPUT

Attendance Log: ['S001', 'S002']
Attendance Log: ['S001']

OBSERVATION

Attendance tracking implemented with deque for efficiency.