ASSIGNMENT 11.3

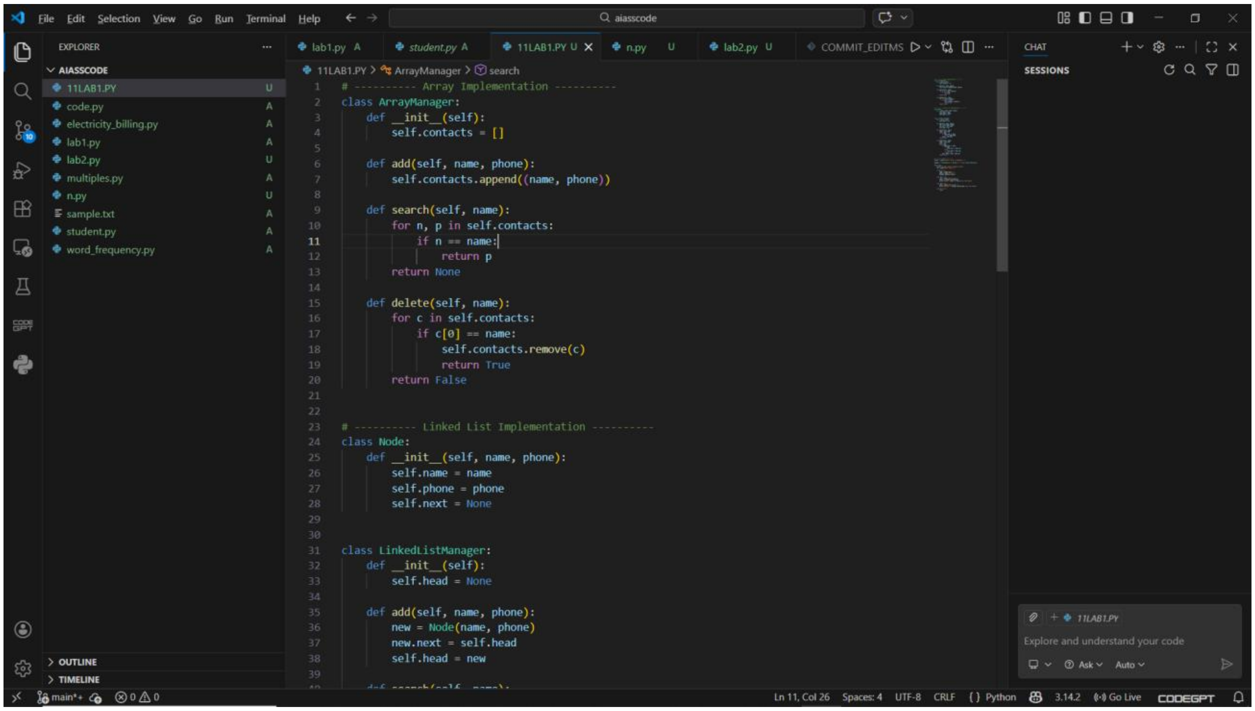
NAME:

HALLTICKET:

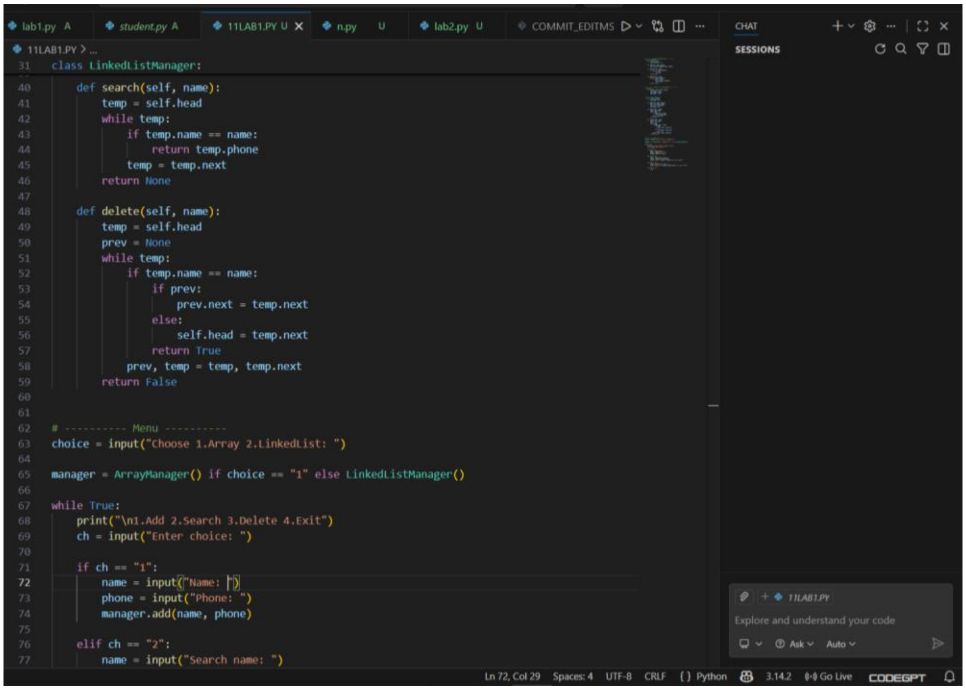
TASK1:

PROMPT

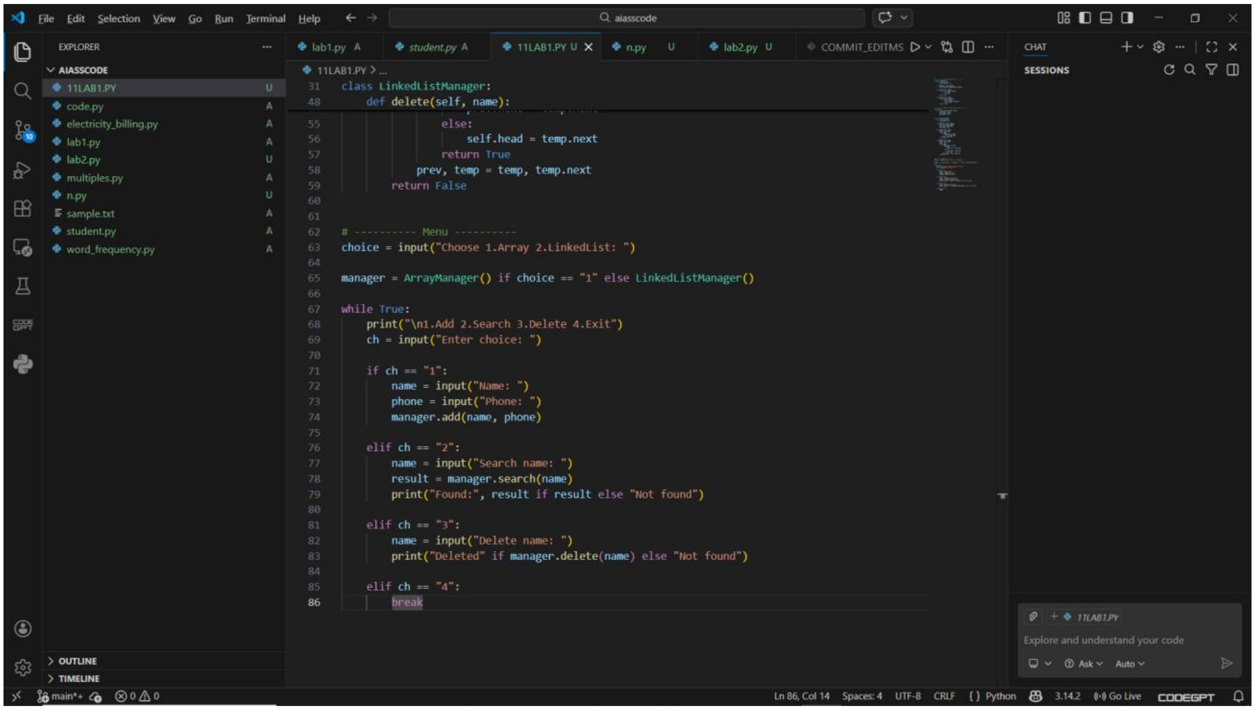
# Create a Smart Contact Manager in Python using both Array (list) and Linked List. Implement operations to add, search, and delete contacts.Provide a simple menu to choose implementation and perform operations.



```python
# ---------- Array Implementation ----------
class ArrayManager:
    def __init__(self):
        self.contacts = []

    def add(self, name, phone):
        self.contacts.append((name, phone))

    def search(self, name):
        for n, p in self.contacts:
            if n == name:
                return p
        return None

    def delete(self, name):
        for c in self.contacts:
            if c[0] == name:
                self.contacts.remove(c)
                return True
        return False


# ---------- Linked List Implementation ----------
class Node:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
        self.next = None


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

    def add(self, name, phone):
        new = Node(name, phone)
        new.next = self.head
        self.head = new

    def search(self, name):
```

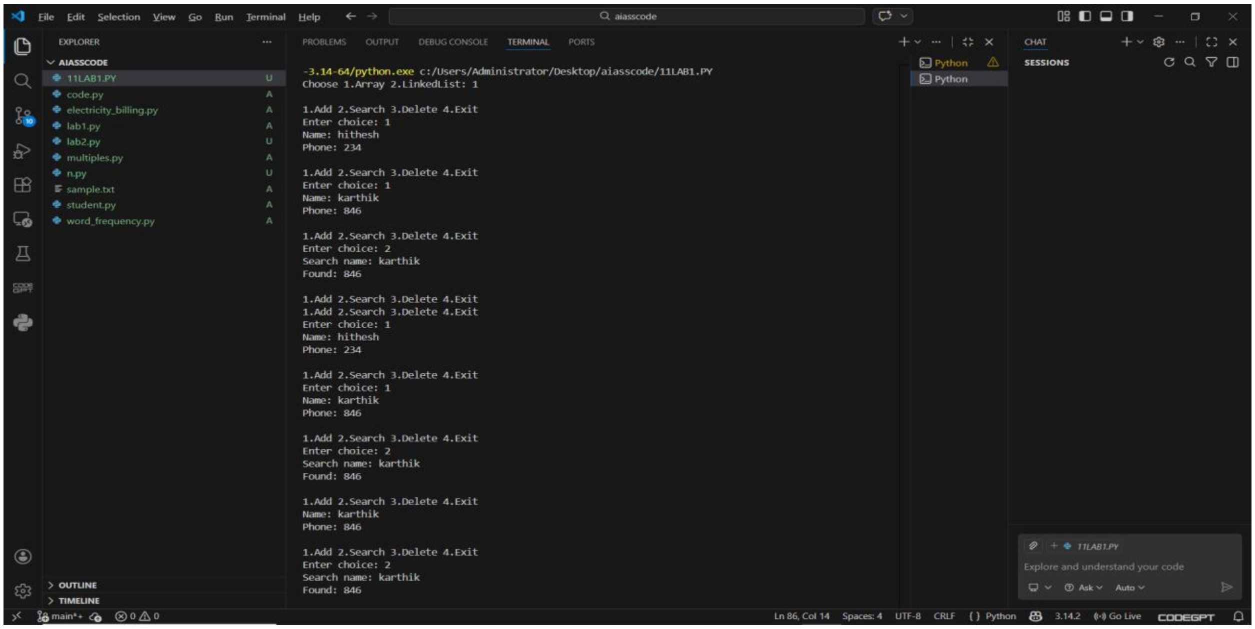


```python
class LinkedListManager:
    def search(self, name):
        temp = self.head
        while temp:
            if temp.name == name:
                return temp.phone
            temp = temp.next
        return None

    def delete(self, name):
        temp = self.head
        prev = None
        while temp:
            if temp.name == name:
                if prev:
                    prev.next = temp.next
                else:
                    self.head = temp.next
                return True
            prev, temp = temp, temp.next
        return False


# ---------- Menu ----------
choice = input("Choose 1.Array 2.LinkedList: ")

manager = ArrayManager() if choice == "1" else LinkedListManager()

while True:
    print("\n1.Add 2.Search 3.Delete 4.Exit")
    ch = input("Enter choice: ")

    if ch == "1":
        name = input("Name: ")
        phone = input("Phone: ")
        manager.add(name, phone)

    elif ch == "2":
        name = input("Search name: ")
```



```python
class LinkedListManager:
    def delete(self, name):
                else:
                    self.head = temp.next
                return True
            prev, temp = temp, temp.next
        return False


# ---------- Menu ----------
choice = input("Choose 1.Array 2.LinkedList: ")

manager = ArrayManager() if choice == "1" else LinkedListManager()

while True:
    print("\n1.Add 2.Search 3.Delete 4.Exit")
    ch = input("Enter choice: ")

    if ch == "1":
        name = input("Name: ")
        phone = input("Phone: ")
        manager.add(name, phone)

    elif ch == "2":
        name = input("Search name: ")
        result = manager.search(name)
        print("Found:", result if result else "Not found")

    elif ch == "3":
        name = input("Delete name: ")
        print("Deleted" if manager.delete(name) else "Not found")

    elif ch == "4":
        break
```

OUTPUT:



```
-3.14-64/python.exe c:/Users/Administrator/Desktop/aiasscode/11LAB1.PY
Choose 1.Array 2.LinkedList: 1

1.Add 2.Search 3.Delete 4.Exit
Enter choice: 1
Name: hithesh
Phone: 234

1.Add 2.Search 3.Delete 4.Exit
Enter choice: 1
Name: karthik
Phone: 846

1.Add 2.Search 3.Delete 4.Exit
Enter choice: 2
Search name: karthik
Found: 846

1.Add 2.Search 3.Delete 4.Exit
1.Add 2.Search 3.Delete 4.Exit
Enter choice: 1
Name: hithesh
Phone: 234

1.Add 2.Search 3.Delete 4.Exit
Enter choice: 1
Name: karthik
Phone: 846

1.Add 2.Search 3.Delete 4.Exit
Enter choice: 2
Search name: karthik
Found: 846

1.Add 2.Search 3.Delete 4.Exit
Name: karthik
Phone: 846

1.Add 2.Search 3.Delete 4.Exit
Enter choice: 2
Search name: karthik
Found: 846
```

Explanation: The array-based contact manager stores contacts in a list, allowing simple insertion but requiring element shifting during deletion. Searching in the array implementation takes linear time as each contact is checked sequentially. The linked list implementation stores contacts using dynamically allocated nodes, allowing efficient insertion at the beginning. Deletion in a linked list is more efficient than arrays since no shifting of elements is required. Both implementations have linear time complexity for searching, but linked lists handle frequent updates better.

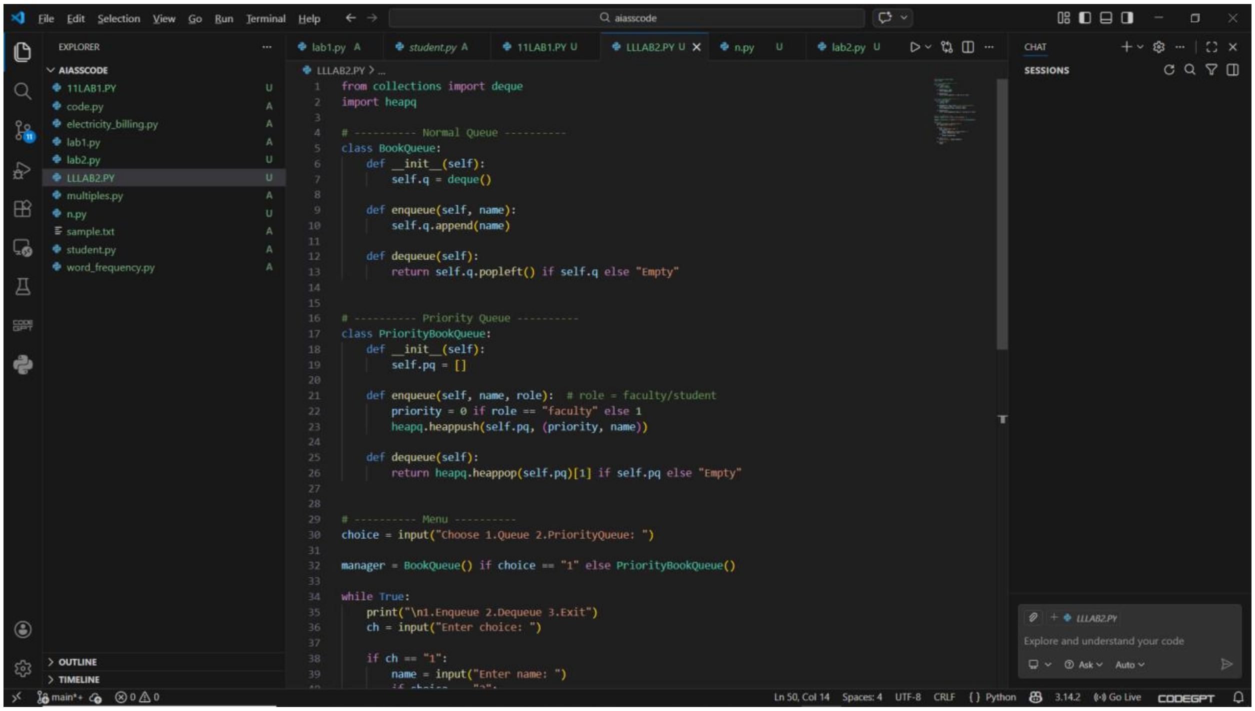
TASK2:

PROMPT:

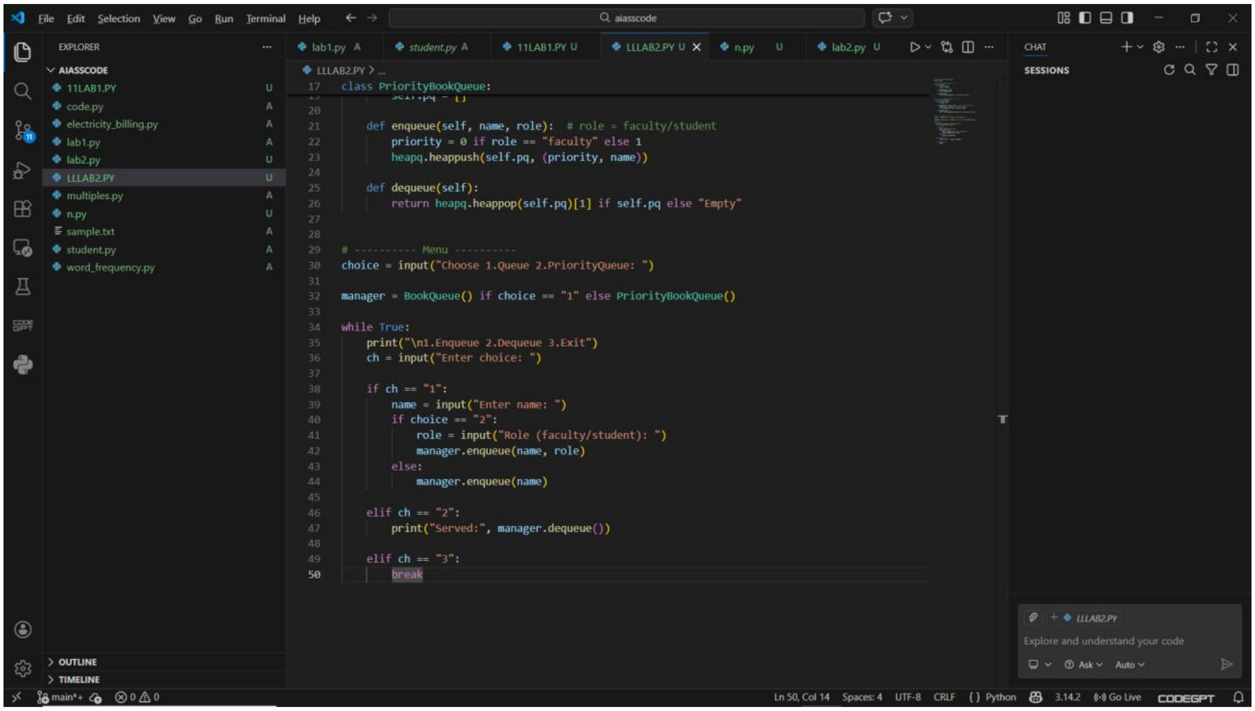
# Create a Library Book Request System in Python.

Implement a normal Queue (FIFO) and a Priority Queue where faculty requests

have higher priority than student requests. Include enqueue and dequeue methods and a simple menu for testing.

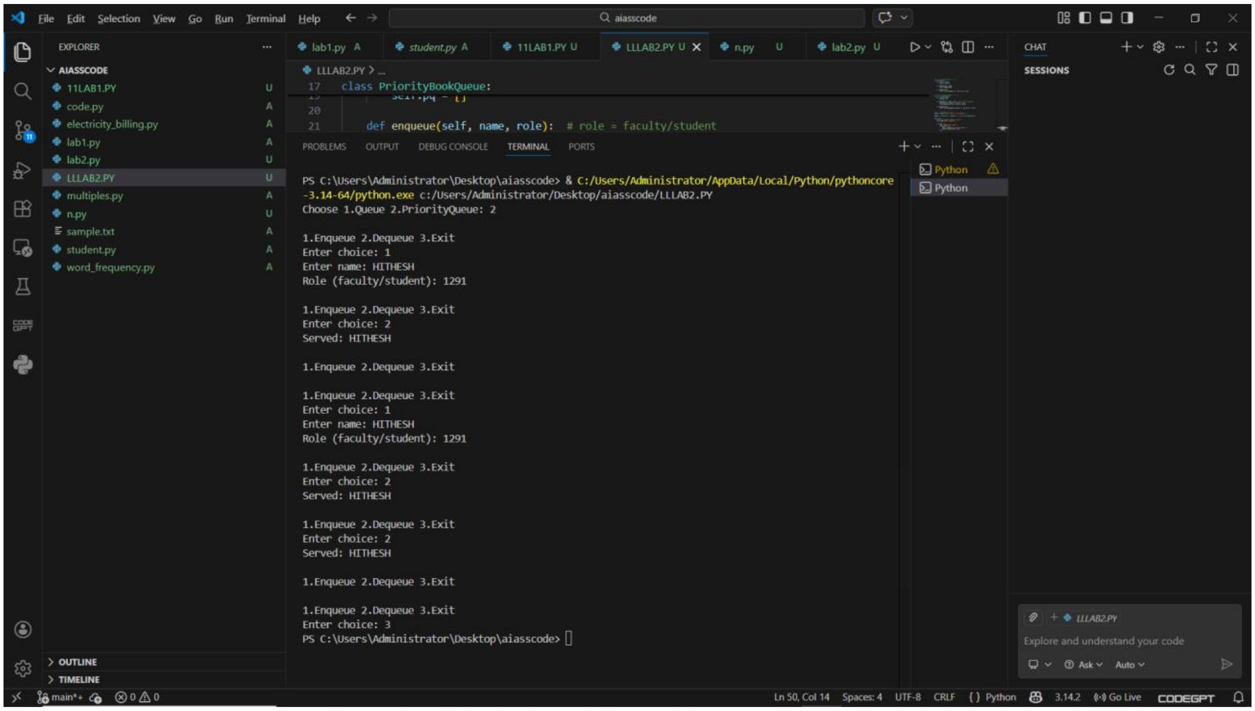


```
from collections import deque
import heapq

# ---------- Normal Queue ----------
class BookQueue:
    def __init__(self):
        self.q = deque()

    def enqueue(self, name):
        self.q.append(name)

    def dequeue(self):
        return self.q.popleft() if self.q else "Empty"


# ---------- Priority Queue ----------
class PriorityBookQueue:
    def __init__(self):
        self.pq = []

    def enqueue(self, name, role):  # role = faculty/student
        priority = 0 if role == "faculty" else 1
        heapq.heappush(self.pq, (priority, name))

    def dequeue(self):
        return heapq.heappop(self.pq)[1] if self.pq else "Empty"


# ---------- Menu ----------
choice = input("Choose 1.Queue 2.PriorityQueue: ")

manager = BookQueue() if choice == "1" else PriorityBookQueue()

while True:
    print("\n1.Enqueue 2.Dequeue 3.Exit")
    ch = input("Enter choice: ")

    if ch == "1":
        name = input("Enter name: ")
```



```
class PriorityBookQueue:

    def enqueue(self, name, role):  # role = faculty/student
        priority = 0 if role == "faculty" else 1
        heapq.heappush(self.pq, (priority, name))

    def dequeue(self):
        return heapq.heappop(self.pq)[1] if self.pq else "Empty"


# ---------- Menu ----------
choice = input("Choose 1.Queue 2.PriorityQueue: ")

manager = BookQueue() if choice == "1" else PriorityBookQueue()

while True:
    print("\n1.Enqueue 2.Dequeue 3.Exit")
    ch = input("Enter choice: ")

    if ch == "1":
        name = input("Enter name: ")
        if choice == "2":
            role = input("Role (faculty/student): ")
            manager.enqueue(name, role)
        else:
            manager.enqueue(name)

    elif ch == "2":
        print("Served:", manager.dequeue())

    elif ch == "3":
        break
```

OUTPUT:



```
class PriorityBookQueue:

    def enqueue(self, name, role):  # role = faculty/student
```

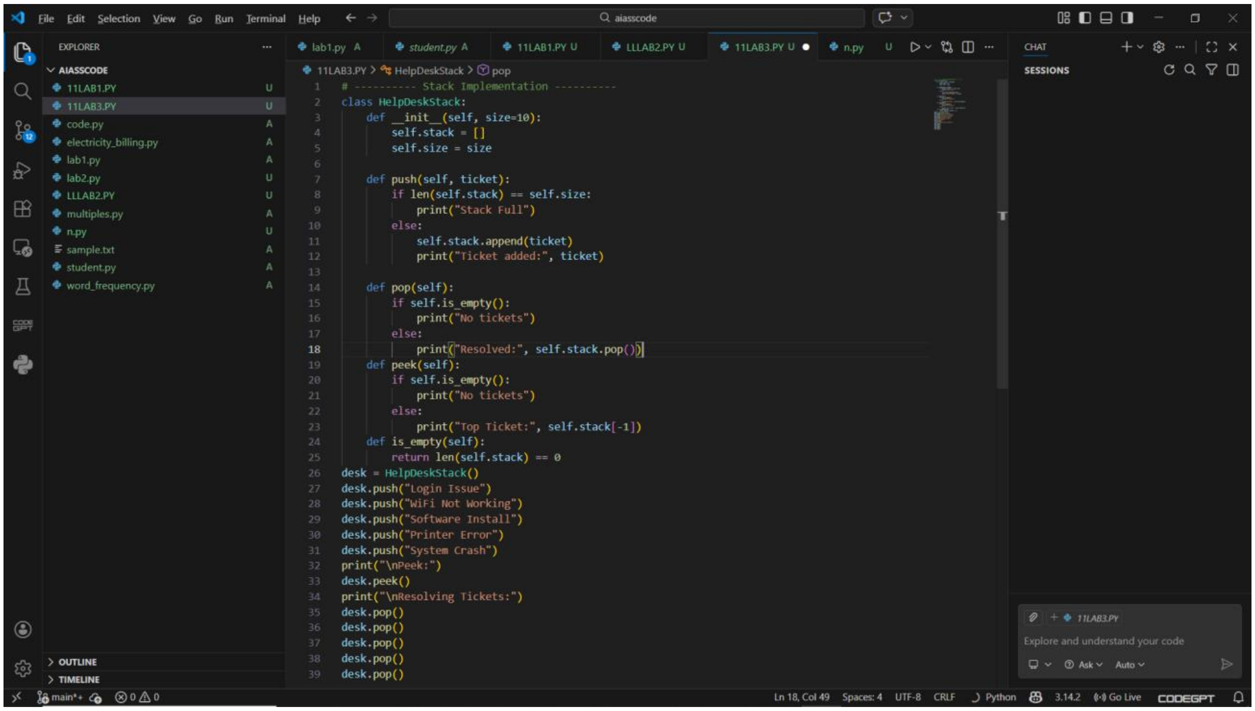
```
PS C:\Users\Administrator\Desktop\aiasscode> & C:/Users/Administrator/AppData/Local/Python/pythoncore
-3.14-64/python.exe c:/Users/Administrator/Desktop/aiasscode/LLLAB2.PY
Choose 1.Queue 2.PriorityQueue: 2

1.Enqueue 2.Dequeue 3.Exit
Enter choice: 1
Enter name: HITHESH
Role (faculty/student): 1291

1.Enqueue 2.Dequeue 3.Exit
Enter choice: 2
Served: HITHESH

1.Enqueue 2.Dequeue 3.Exit

1.Enqueue 2.Dequeue 3.Exit
Enter choice: 1
Enter name: HITHESH
Role (faculty/student): 1291

1.Enqueue 2.Dequeue 3.Exit
Enter choice: 2
Served: HITHESH

1.Enqueue 2.Dequeue 3.Exit
Enter choice: 2
Served: HITHESH

1.Enqueue 2.Dequeue 3.Exit

1.Enqueue 2.Dequeue 3.Exit
Enter choice: 3
PS C:\Users\Administrator\Desktop\aiasscode> 
```

Explanation: This program implements a library book request system using a normal queue and a priority queue. The normal queue follows FIFO order and processes requests in the order they arrive. The priority queue separates faculty and student requests and always serves faculty first. The enqueue method adds requests, while the dequeue method removes the correct request based on priority. This approach demonstrates how priority queues handle real-world situations where some requests are more important

TASK3:

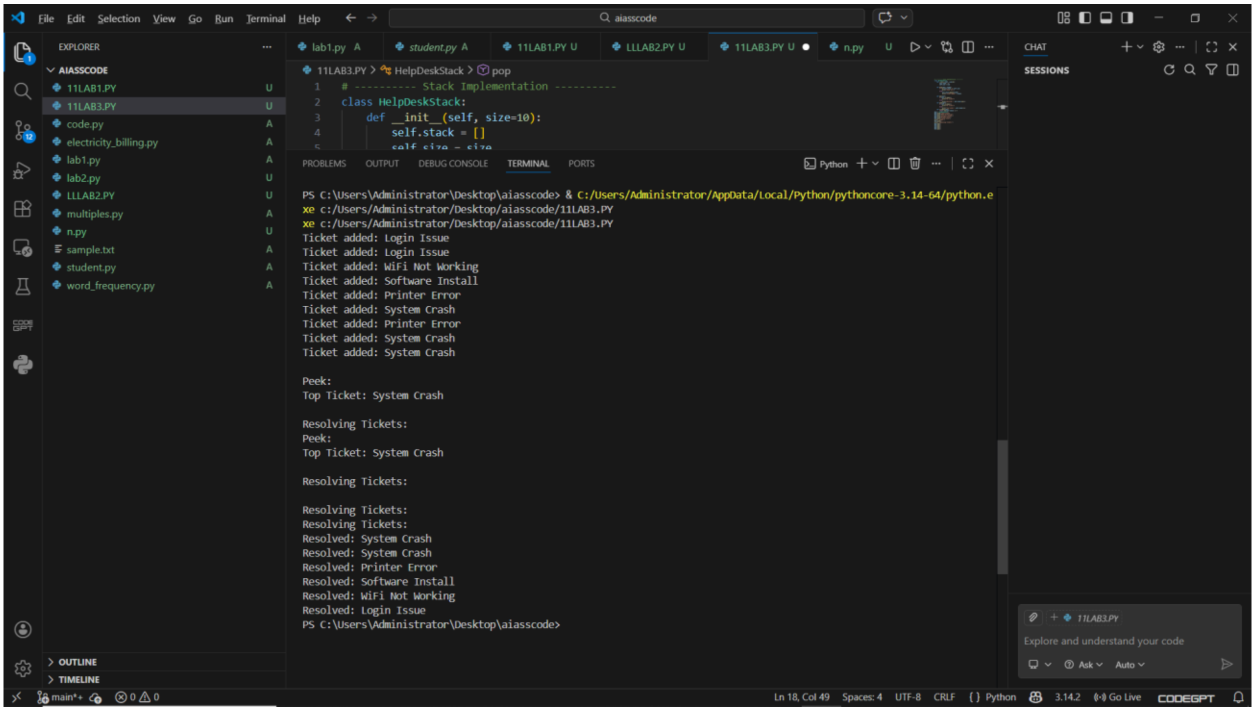
PROMPT:

# Create an Emergency Help Desk Ticket System using Stack in Python.Implement push, pop, peek operations and methods to check if stack is empty or full.Simulate at least five tickets being raised and resolved to demonstrate LIFO behavior.



```python
# ---------- Stack Implementation ----------
class HelpDeskStack:
    def __init__(self, size=10):
        self.stack = []
        self.size = size

    def push(self, ticket):
        if len(self.stack) == self.size:
            print("Stack Full")
        else:
            self.stack.append(ticket)
            print("Ticket added:", ticket)

    def pop(self):
        if self.is_empty():
            print("No tickets")
        else:
            print("Resolved:", self.stack.pop())
    def peek(self):
        if self.is_empty():
            print("No tickets")
        else:
            print("Top Ticket:", self.stack[-1])
    def is_empty(self):
        return len(self.stack) == 0
desk = HelpDeskStack()
desk.push("Login Issue")
desk.push("WiFi Not Working")
desk.push("Software Install")
desk.push("Printer Error")
desk.push("System Crash")
print("\nPeek:")
desk.peek()
print("\nResolving Tickets:")
desk.pop()
desk.pop()
desk.pop()
desk.pop()
desk.pop()
```

OUTPUT:



```python
# ---------- Stack Implementation ----------
class HelpDeskStack:
    def __init__(self, size=10):
        self.stack = []
```

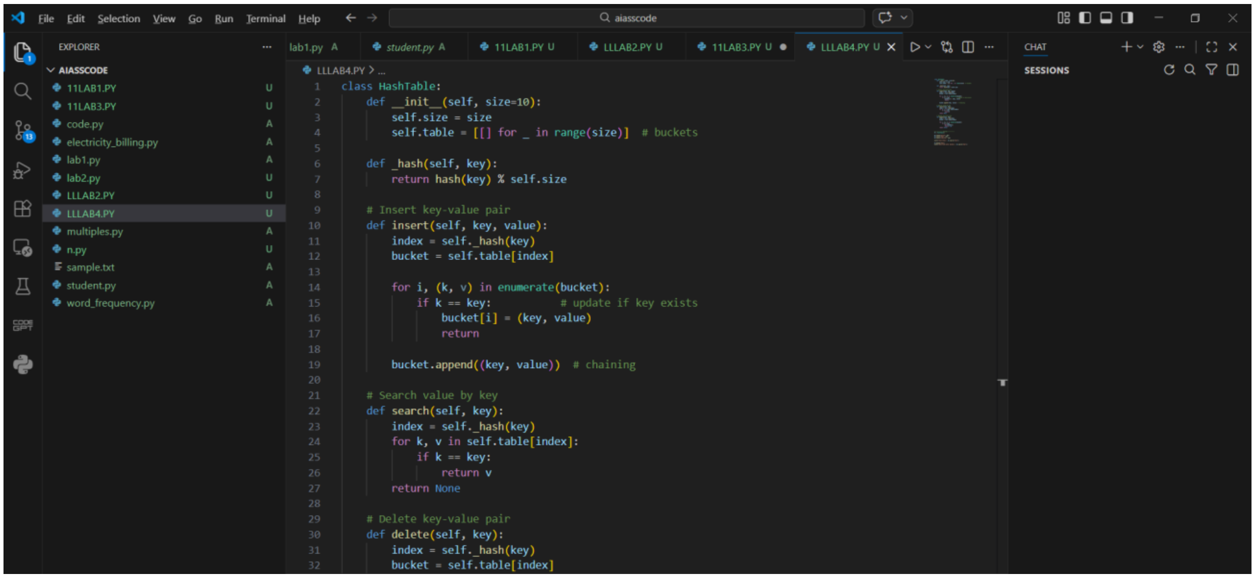
```
PS C:\Users\Administrator\Desktop\aiasscode> & C:/Users/Administrator/AppData/Local/Python/pythoncore-3.14-64/python.e
xe c:/Users/Administrator/Desktop/aiasscode/11LAB3.PY
xe c:/Users/Administrator/Desktop/aiasscode/11LAB3.PY
Ticket added: Login Issue
Ticket added: Login Issue
Ticket added: WiFi Not Working
Ticket added: Software Install
Ticket added: Printer Error
Ticket added: System Crash
Ticket added: Printer Error
Ticket added: System Crash
Ticket added: System Crash

Peek:
Top Ticket: System Crash

Resolving Tickets:
Peek:
Top Ticket: System Crash

Resolving Tickets:

Resolving Tickets:
Resolving Tickets:
Resolved: System Crash
Resolved: System Crash
Resolved: Printer Error
Resolved: Software Install
Resolved: WiFi Not Working
Resolved: Login Issue
PS C:\Users\Administrator\Desktop\aiasscode>
```

Explanation: This program models an IT help desk using a stack, where support tickets are handled in Last-In, First-Out order. Each ticket stores an ID, requester name, and issue description, and is added to the stack using the push operation. The pop operation resolves the most recently added ticket first, clearly showing LIFO behavior. Additional methods like peek, is_empty, and is_full help manage and check the stack safely

TASK4:

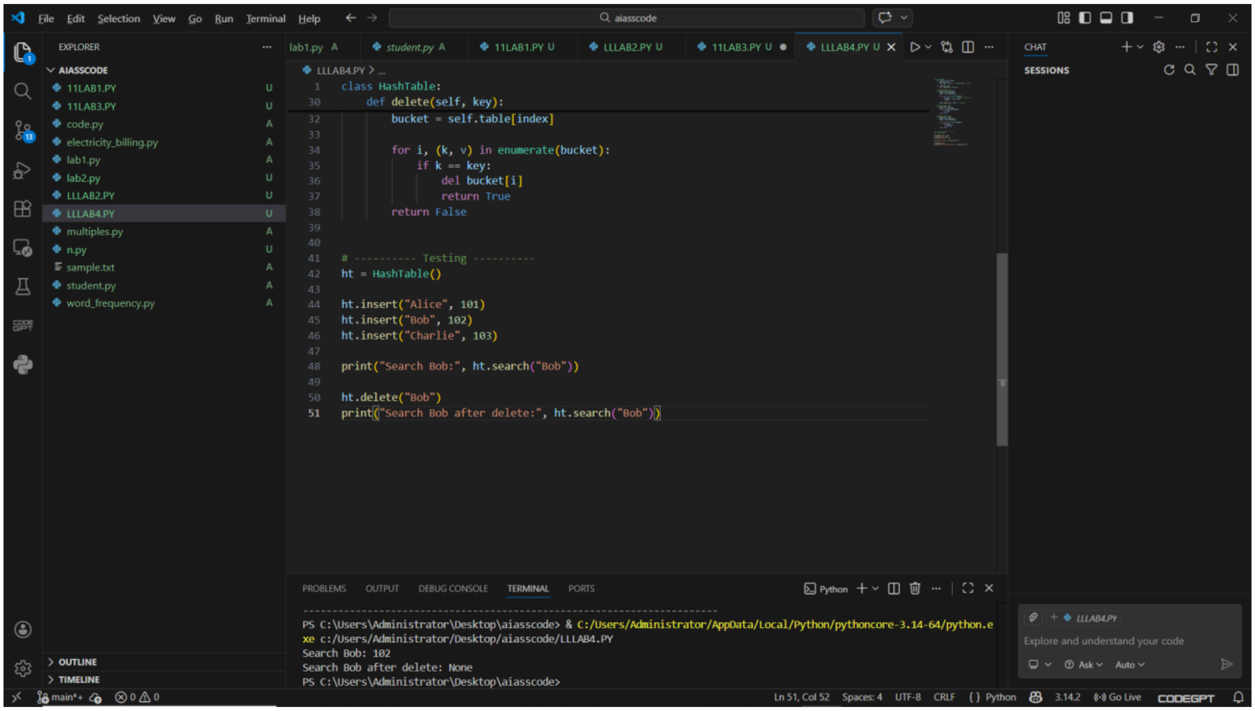
PROMPT:

# Create a Hash Table in Python using chaining for collision handling. Implement insert, search, and delete methods with proper comments.Test the hash table with some sample data.



```python
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]  # buckets

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

    # Insert key-value pair
    def insert(self, key, value):
        index = self._hash(key)
        bucket = self.table[index]

        for i, (k, v) in enumerate(bucket):
            if k == key:            # update if key exists
                bucket[i] = (key, value)
                return

        bucket.append((key, value))  # chaining

    # Search value by key
    def search(self, key):
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None

    # Delete key-value pair
    def delete(self, key):
        index = self._hash(key)
        bucket = self.table[index]
```

OUTPUT:



```python
class HashTable:
    def delete(self, key):
        bucket = self.table[index]

        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                return True
        return False


# ---------- Testing ----------
ht = HashTable()

ht.insert("Alice", 101)
ht.insert("Bob", 102)
ht.insert("Charlie", 103)

print("Search Bob:", ht.search("Bob"))

ht.delete("Bob")
print("Search Bob after delete:", ht.search("Bob"))
```

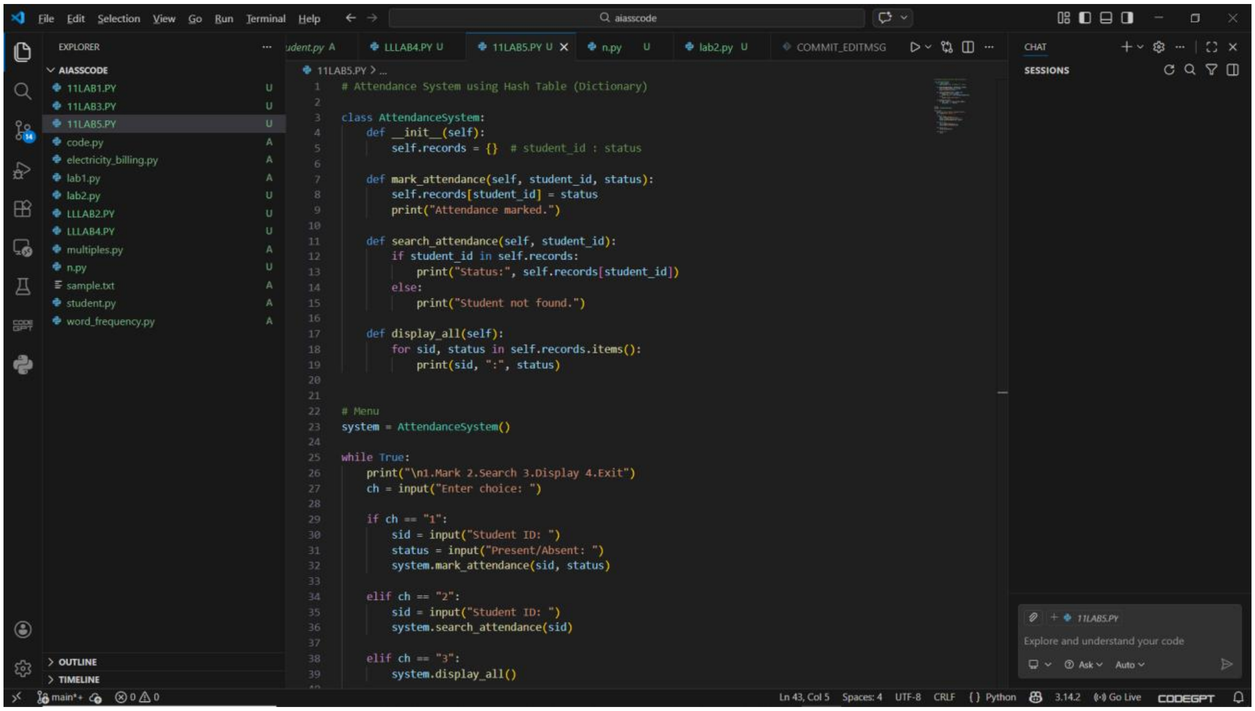
```
---------------------------------------------------------------------------
PS C:\Users\Administrator\Desktop\aiasscode> & C:/Users/Administrator/AppData/Local/Python/pythoncore-3.14-64/python.e
xe c:/Users/Administrator/Desktop/aiasscode/LLLAB4.PY
Search Bob: 102
Search Bob after delete: None
PS C:\Users\Administrator\Desktop\aiasscode>
```

Explanation: This program implements a hash table using the chaining method to handle collisions. Each table index stores a linked list of nodes, allowing multiple keys to share the same index safely. The insert() method places a key–value pair into the appropriate bucket or updates it if the key already exists. The search() method traverses the linked list at the computed index to find the required key. The delete() method removes the node without breaking the chain, ensuring correct hash table behavior

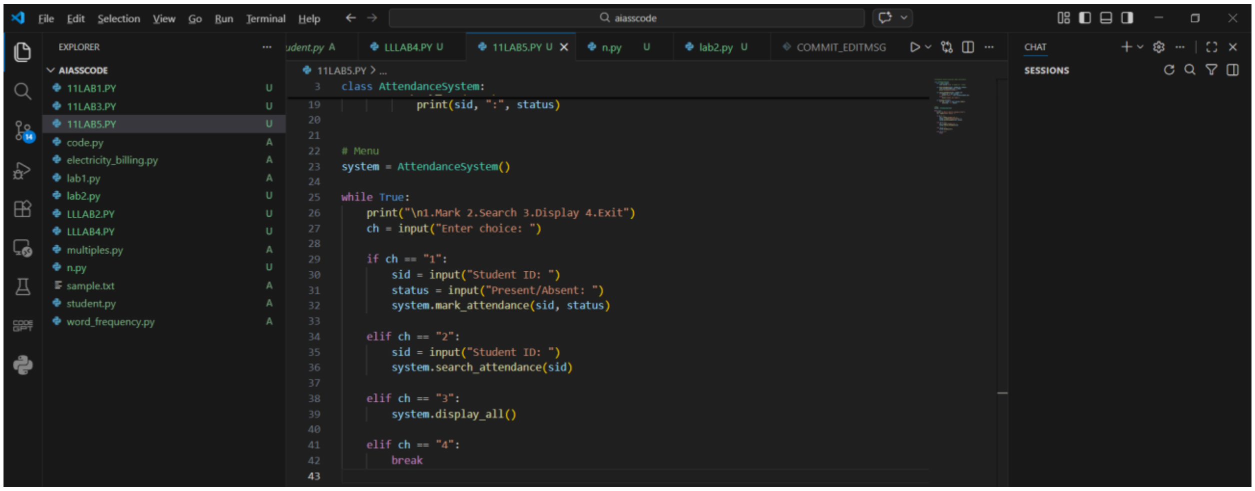
TASK5:

PROMPT:

# Create a Student Attendance Tracking System using a dictionary (hash table). Implement functions to mark attendance, search attendance, and display all records. Use a simple menu-driven program.

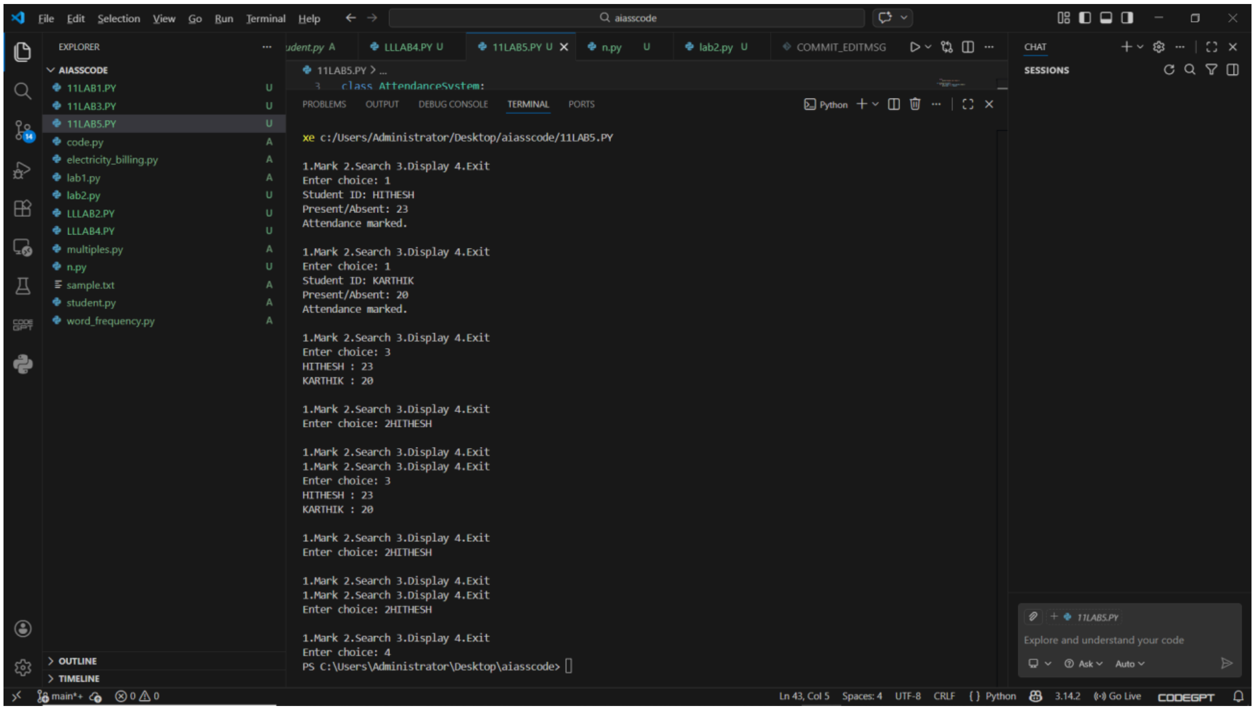


```python
# Attendance System using Hash Table (Dictionary)

class AttendanceSystem:
    def __init__(self):
        self.records = {}  # student_id : status

    def mark_attendance(self, student_id, status):
        self.records[student_id] = status
        print("Attendance marked.")

    def search_attendance(self, student_id):
        if student_id in self.records:
            print("Status:", self.records[student_id])
        else:
            print("Student not found.")

    def display_all(self):
        for sid, status in self.records.items():
            print(sid, ":", status)


# Menu
system = AttendanceSystem()

while True:
    print("\n1.Mark 2.Search 3.Display 4.Exit")
    ch = input("Enter choice: ")

    if ch == "1":
        sid = input("Student ID: ")
        status = input("Present/Absent: ")
        system.mark_attendance(sid, status)

    elif ch == "2":
        sid = input("Student ID: ")
        system.search_attendance(sid)

    elif ch == "3":
        system.display_all()
```



```
class AttendanceSystem:
            print(sid, ":", status)


# Menu
system = AttendanceSystem()

while True:
    print("\n1.Mark 2.Search 3.Display 4.Exit")
    ch = input("Enter choice: ")

    if ch == "1":
        sid = input("Student ID: ")
        status = input("Present/Absent: ")
        system.mark_attendance(sid, status)

    elif ch == "2":
        sid = input("Student ID: ")
        system.search_attendance(sid)

    elif ch == "3":
        system.display_all()

    elif ch == "4":
        break
```

OUTPUT:



```
xe c:/Users/Administrator/Desktop/aiasscode/11LAB5.PY

1.Mark 2.Search 3.Display 4.Exit
Enter choice: 1
Student ID: HITHESH
Present/Absent: 23
Attendance marked.

1.Mark 2.Search 3.Display 4.Exit
Enter choice: 1
Student ID: KARTHIK
Present/Absent: 20
Attendance marked.

1.Mark 2.Search 3.Display 4.Exit
Enter choice: 3
HITHESH : 23
KARTHIK : 20

1.Mark 2.Search 3.Display 4.Exit
Enter choice: 2HITHESH

1.Mark 2.Search 3.Display 4.Exit
1.Mark 2.Search 3.Display 4.Exit
Enter choice: 3
HITHESH : 23
KARTHIK : 20

1.Mark 2.Search 3.Display 4.Exit
Enter choice: 2HITHESH

1.Mark 2.Search 3.Display 4.Exit
1.Mark 2.Search 3.Display 4.Exit
Enter choice: 2HITHESH

1.Mark 2.Search 3.Display 4.Exit
Enter choice: 4
PS C:\Users\Administrator\Desktop\aiasscode> 
```

Explanation: I selected the Library Book Borrowing System because it requires fast access, insertion, and deletion of book records. A hash table is used where the key is book_id and the value is book_name. Adding a book stores the key–value pair directly, giving O(1) average time complexity. Removing a book is efficient because the book ID allows direct lookup without traversal