AI Assisted Coding

Assignment 11.3

Name: R.Bharadwaj

Hall ticket no: 2303A51610

Batch no: 19

Task 1: Smart Contact Manager (Arrays & Linked Lists)

Prompt:

Generate Python code to implement a Contact Manager system using:

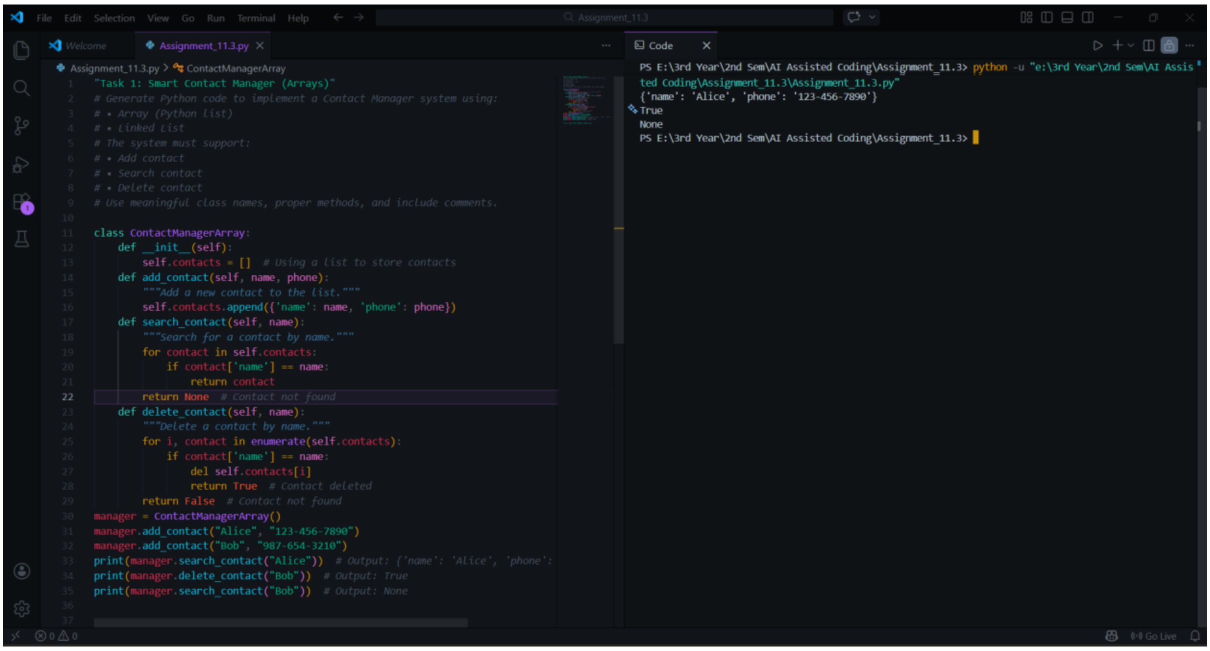
- Array (Python list)
- Linked List

The system must support:

- Add contact
- Search contact
- Delete contact

Use meaningful class names, proper methods, and include comments.

Code & Output (Arrays):



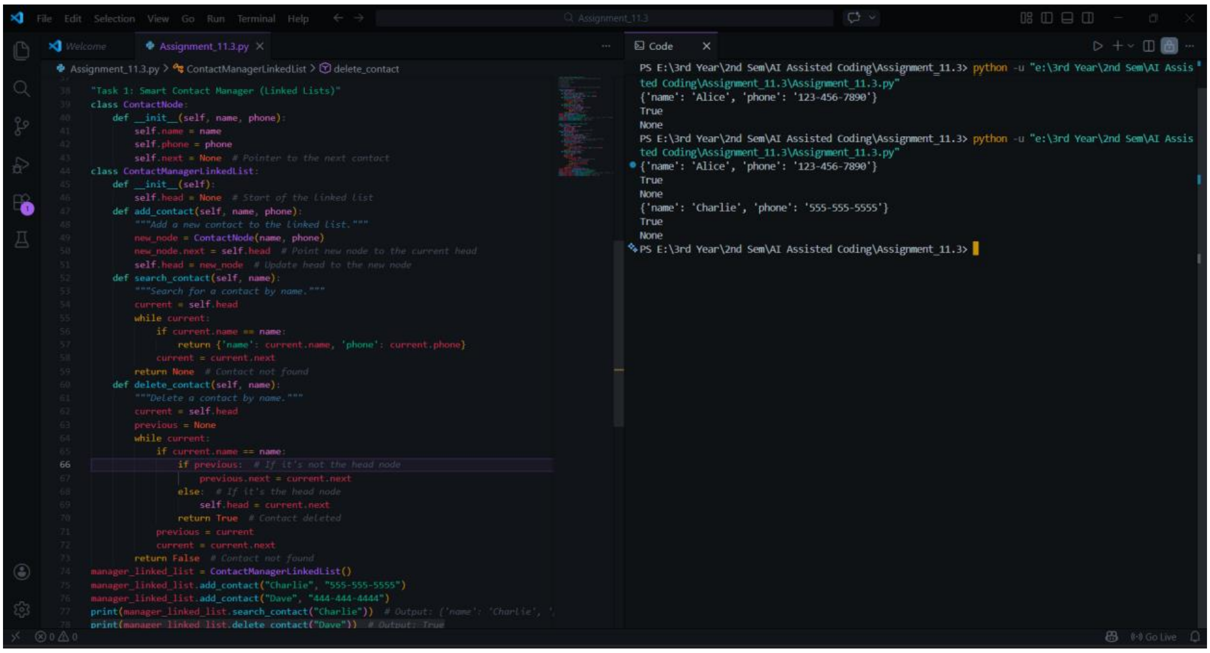
```python
"Task 1: Smart Contact Manager (Arrays)"
# Generate Python code to implement a Contact Manager system using:
# • Array (Python list)
# • Linked List
# The system must support:
# • Add contact
# • Search contact
# • Delete contact
# Use meaningful class names, proper methods, and include comments.

class ContactManagerArray:
    def __init__(self):
        self.contacts = []  # Using a list to store contacts
    def add_contact(self, name, phone):
        """Add a new contact to the list."""
        self.contacts.append({'name': name, 'phone': phone})
    def search_contact(self, name):
        """Search for a contact by name."""
        for contact in self.contacts:
            if contact['name'] == name:
                return contact
        return None  # Contact not found
    def delete_contact(self, name):
        """Delete a contact by name."""
        for i, contact in enumerate(self.contacts):
            if contact['name'] == name:
                del self.contacts[i]
                return True  # Contact deleted
        return False  # Contact not found
manager = ContactManagerArray()
manager.add_contact("Alice", "123-456-7890")
manager.add_contact("Bob", "987-654-3210")
print(manager.search_contact("Alice"))  # Output: {'name': 'Alice', 'phone':
print(manager.delete_contact("Bob"))  # Output: True
print(manager.search_contact("Bob"))  # Output: None
```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python -u "e:\3rd Year\2nd Sem\AI Assis
ted Coding\Assignment_11.3\Assignment_11.3.py"
{'name': 'Alice', 'phone': '123-456-7890'}
True
None
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

Explanation (Arrays):

This implementation uses a Python list to store contact dictionaries. Adding contacts is efficient (O(1) average). Searching and deletion require linear traversal (O(n)). The array approach is simple and easy to implement but less efficient for frequent deletions in large datasets.

Code & Output (Linked-Lists):



```python
"Task 1: Smart Contact Manager (Linked Lists)"
class ContactNode:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
        self.next = None  # Pointer to the next contact
class ContactManagerLinkedList:
    def __init__(self):
        self.head = None  # Start of the linked list
    def add_contact(self, name, phone):
        """Add a new contact to the linked list."""
        new_node = ContactNode(name, phone)
        new_node.next = self.head  # Point new node to the current head
        self.head = new_node  # Update head to the new node
    def search_contact(self, name):
        """Search for a contact by name."""
        current = self.head
        while current:
            if current.name == name:
                return {'name': current.name, 'phone': current.phone}
            current = current.next
        return None  # Contact not found
    def delete_contact(self, name):
        """Delete a contact by name."""
        current = self.head
        previous = None
        while current:
            if current.name == name:
                if previous:  # If it's not the head node
                    previous.next = current.next
                else:  # If it's the head node
                    self.head = current.next
                return True  # Contact deleted
            previous = current
            current = current.next
        return False  # Contact not found
manager_linked_list = ContactManagerLinkedList()
manager_linked_list.add_contact("Charlie", "555-555-5555")
manager_linked_list.add_contact("Dave", "444-444-4444")
print(manager_linked_list.search_contact("Charlie"))  # Output: {'name': 'Charlie', '
print(manager_linked_list.delete_contact("Dave"))  # Output: True
```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python -u "e:\3rd Year\2nd Sem\AI Assis
ted Coding\Assignment_11.3\Assignment_11.3.py"
{'name': 'Alice', 'phone': '123-456-7890'}
True
None
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python -u "e:\3rd Year\2nd Sem\AI Assis
ted Coding\Assignment_11.3\Assignment_11.3.py"
{'name': 'Alice', 'phone': '123-456-7890'}
True
None
{'name': 'Charlie', 'phone': '555-555-5555'}
True
None
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

Explanation (Linked-Lists):

The linked list implementation allows dynamic memory allocation. Insertion at the beginning is O(1). Searching and deletion are O(n). Unlike arrays, linked lists avoid shifting elements during deletion. However, they require extra memory for pointers and are slightly more complex to implement.

Comparision (Arrays VS Linked-Lists):

- Insertion Efficiency: Linked List (O(1) at head) is better than array when frequent insertions occur.
- Deletion Efficiency: Linked List avoids shifting elements.
- Search Efficiency: Both require O(n).
- Memory Usage: Array is more memory-efficient.

Task 2: Library Book Search System (Queue & Priority Queue)

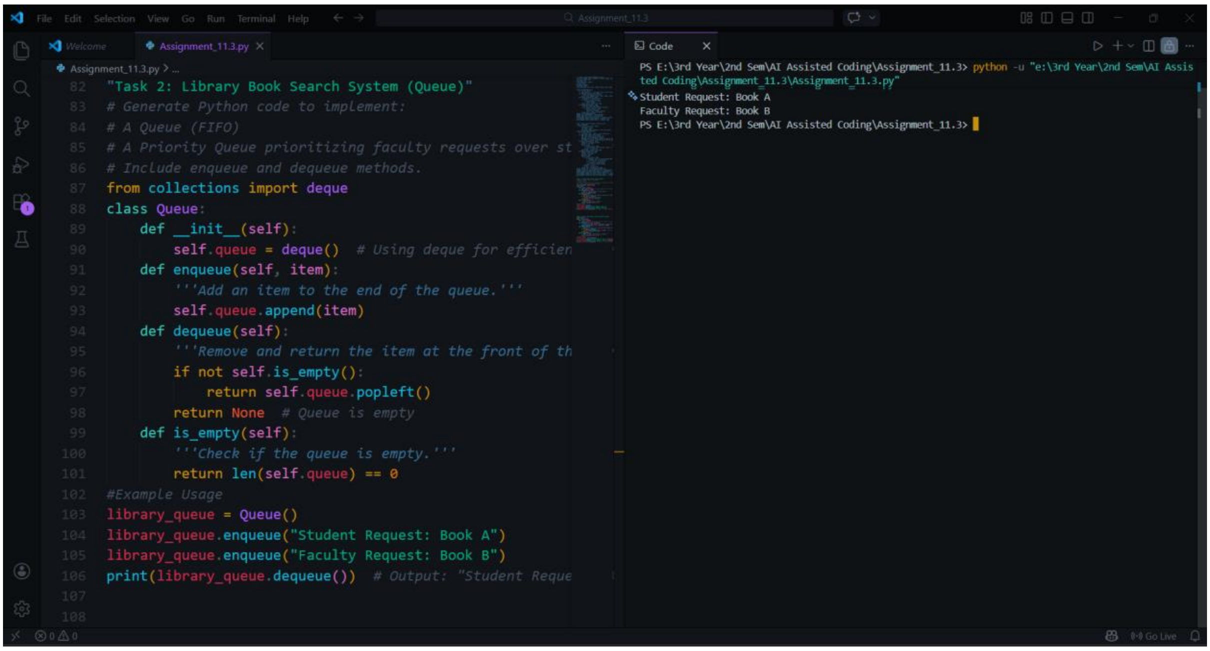
Prompt:

Generate Python code to implement:

- A Queue (FIFO)
- A Priority Queue prioritizing faculty requests over student requests

Include enqueue and dequeue methods.

Code & Output (Queue):



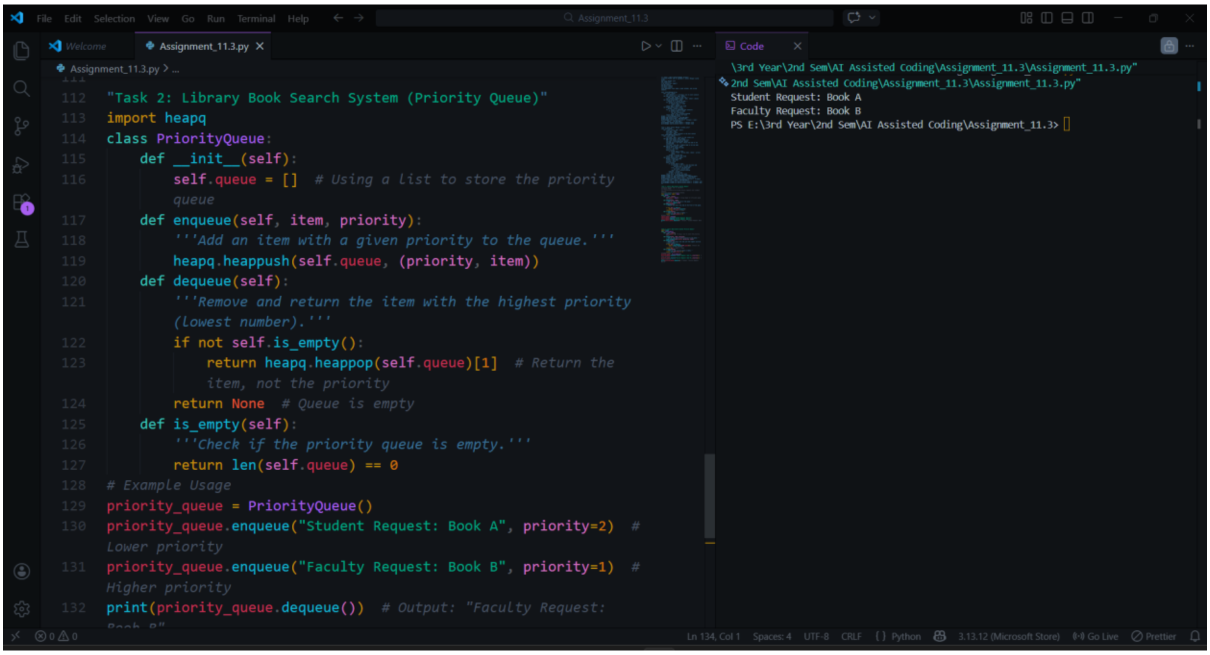
```python
"Task 2: Library Book Search System (Queue)"
# Generate Python code to implement:
# A Queue (FIFO)
# A Priority Queue prioritizing faculty requests over st
# Include enqueue and dequeue methods.
from collections import deque
class Queue:
    def __init__(self):
        self.queue = deque()  # Using deque for efficien
    def enqueue(self, item):
        '''Add an item to the end of the queue.'''
        self.queue.append(item)
    def dequeue(self):
        '''Remove and return the item at the front of th
        if not self.is_empty():
            return self.queue.popleft()
        return None  # Queue is empty
    def is_empty(self):
        '''Check if the queue is empty.'''
        return len(self.queue) == 0
#Example Usage
library_queue = Queue()
library_queue.enqueue("Student Request: Book A")
library_queue.enqueue("Faculty Request: Book B")
print(library_queue.dequeue())  # Output: "Student Reque
```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python -u "e:\3rd Year\2nd Sem\AI Assis
ted Coding\Assignment_11.3\Assignment_11.3.py"
Student Request: Book A
Faculty Request: Book B
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

Explanation (Queue):

The queue follows FIFO (First In, First Out). Requests are processed in the order they arrive. This is suitable for standard book request management.

Code & Output (Priority Queue):



```python
"Task 2: Library Book Search System (Priority Queue)"
import heapq
class PriorityQueue:
    def __init__(self):
        self.queue = []  # Using a list to store the priority queue
    def enqueue(self, item, priority):
        '''Add an item with a given priority to the queue.'''
        heapq.heappush(self.queue, (priority, item))
    def dequeue(self):
        '''Remove and return the item with the highest priority (lowest number).'''
        if not self.is_empty():
            return heapq.heappop(self.queue)[1]  # Return the item, not the priority
        return None  # Queue is empty
    def is_empty(self):
        '''Check if the priority queue is empty.'''
        return len(self.queue) == 0
# Example Usage
priority_queue = PriorityQueue()
priority_queue.enqueue("Student Request: Book A", priority=2)  # Lower priority
priority_queue.enqueue("Faculty Request: Book B", priority=1)  # Higher priority
print(priority_queue.dequeue())  # Output: "Faculty Request: Book B"
```

```
\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3\Assignment_11.3.py"
2nd Sem\AI Assisted Coding\Assignment_11.3\Assignment_11.3.py"
Student Request: Book A
Faculty Request: Book B
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

Explanation (Priority Queue):

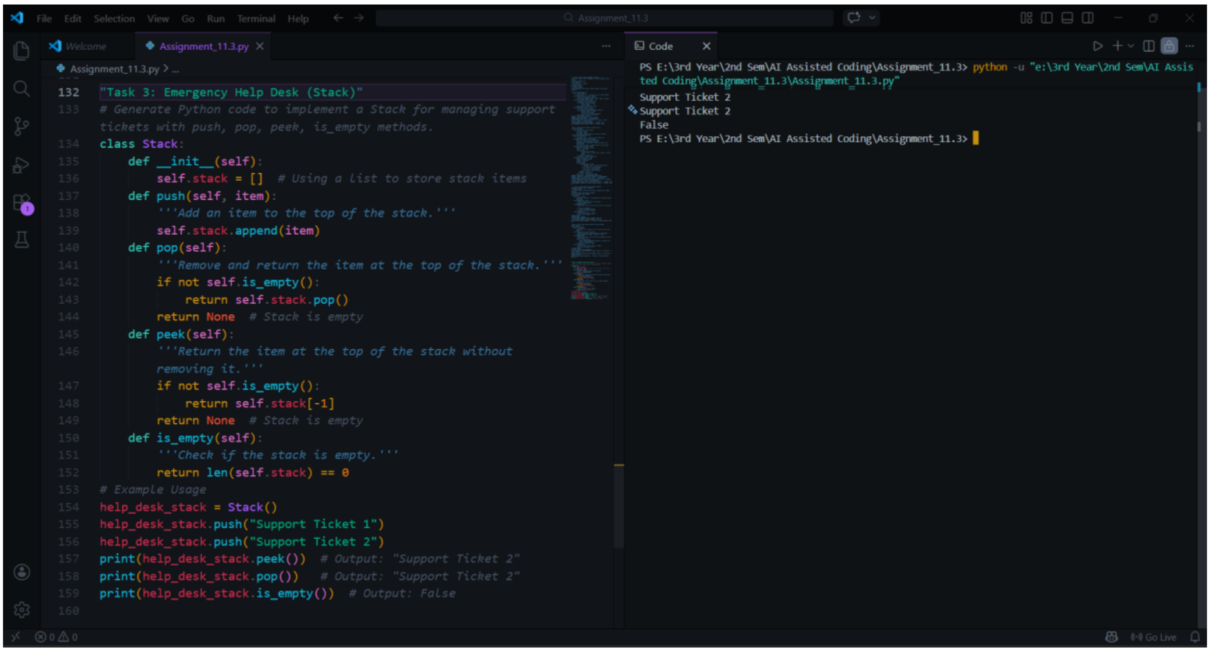
The priority queue uses a heap. Faculty requests are assigned higher priority (lower numeric value). This ensures faculty members are served before students.

Task 3: Emergency Help Desk (Stack)

Prompt:

Generate Python code to implement a Stack for managing support tickets with push, pop, peek, is_empty methods.

Code & Output:



```python
"Task 3: Emergency Help Desk (Stack)"
# Generate Python code to implement a Stack for managing support tickets with push, pop, peek, is_empty methods.
class Stack:
    def __init__(self):
        self.stack = []  # Using a list to store stack items
    def push(self, item):
        '''Add an item to the top of the stack.'''
        self.stack.append(item)
    def pop(self):
        '''Remove and return the item at the top of the stack.'''
        if not self.is_empty():
            return self.stack.pop()
        return None  # Stack is empty
    def peek(self):
        '''Return the item at the top of the stack without removing it.'''
        if not self.is_empty():
            return self.stack[-1]
        return None  # Stack is empty
    def is_empty(self):
        '''Check if the stack is empty.'''
        return len(self.stack) == 0
# Example Usage
help_desk_stack = Stack()
help_desk_stack.push("Support Ticket 1")
help_desk_stack.push("Support Ticket 2")
print(help_desk_stack.peek())  # Output: "Support Ticket 2"
print(help_desk_stack.pop())   # Output: "Support Ticket 2"
print(help_desk_stack.is_empty())  # Output: False
```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3\Assignment_11.3.py"
Support Ticket 2
Support Ticket 2
False
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

Explanation:

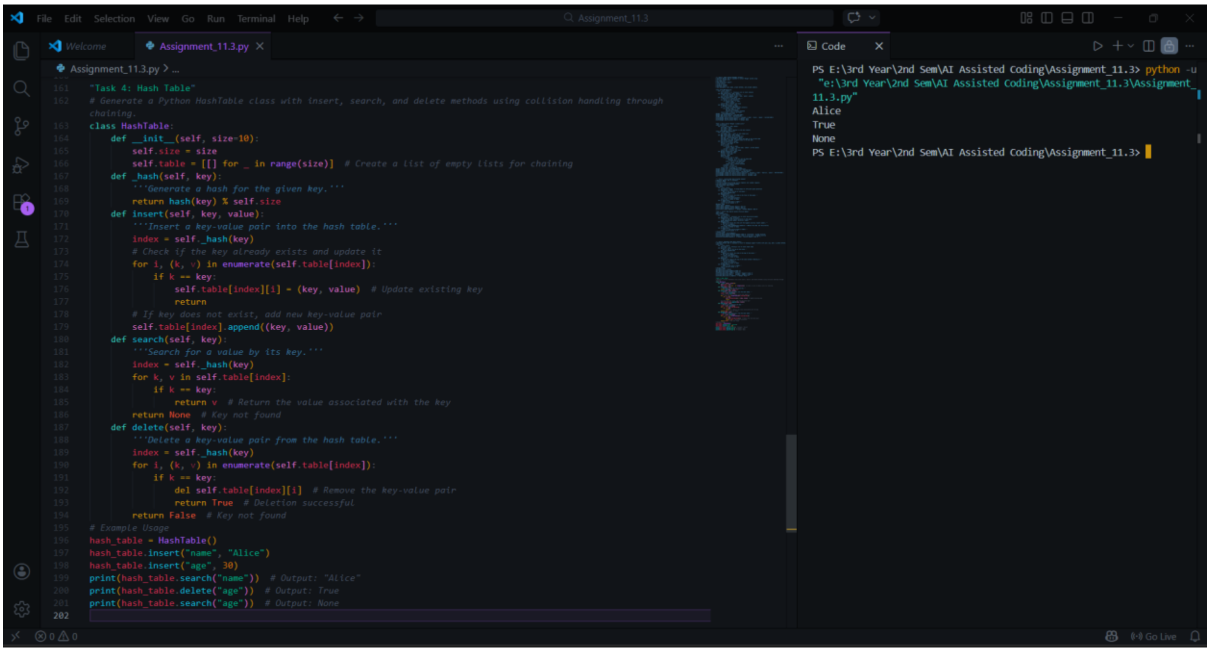
The stack manages support tickets using LIFO order, where the most recent ticket is resolved first. Push, pop, and peek operations demonstrate escalation handling effectively. This structure is suitable for urgent issue resolution workflows. AI assistance helped design stack methods and improve operational clarity.

Task 4: Hash Table

Prompt:

Generate a Python HashTable class with insert, search, and delete methods using collision handling through chaining.

Code & Output:



```python
"Task 4: Hash Table"
# Generate a Python HashTable class with insert, search, and delete methods using collision handling through chaining.
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]  # Create a list of empty lists for chaining
    def _hash(self, key):
        '''Generate a hash for the given key.'''
        return hash(key) % self.size
    def insert(self, key, value):
        '''Insert a key-value pair into the hash table.'''
        index = self._hash(key)
        # Check if the key already exists and update it
        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
        self.table[index].append((key, value))
    def search(self, key):
        '''Search for a value by its key.'''
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v  # Return the value associated with the key
        return None  # Key not found
    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]):
            if k == key:
                del self.table[index][i]  # Remove the key-value pair
                return True  # Deletion successful
        return False  # Key not found
# Example Usage
hash_table = HashTable()
hash_table.insert("name", "Alice")
hash_table.insert("age", 30)
print(hash_table.search("name"))  # Output: "Alice"
print(hash_table.delete("age"))  # Output: True
print(hash_table.search("age"))  # Output: None
```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python -u
 "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3\Assignment_
11.3.py"
Alice
True
None
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

Explanation:

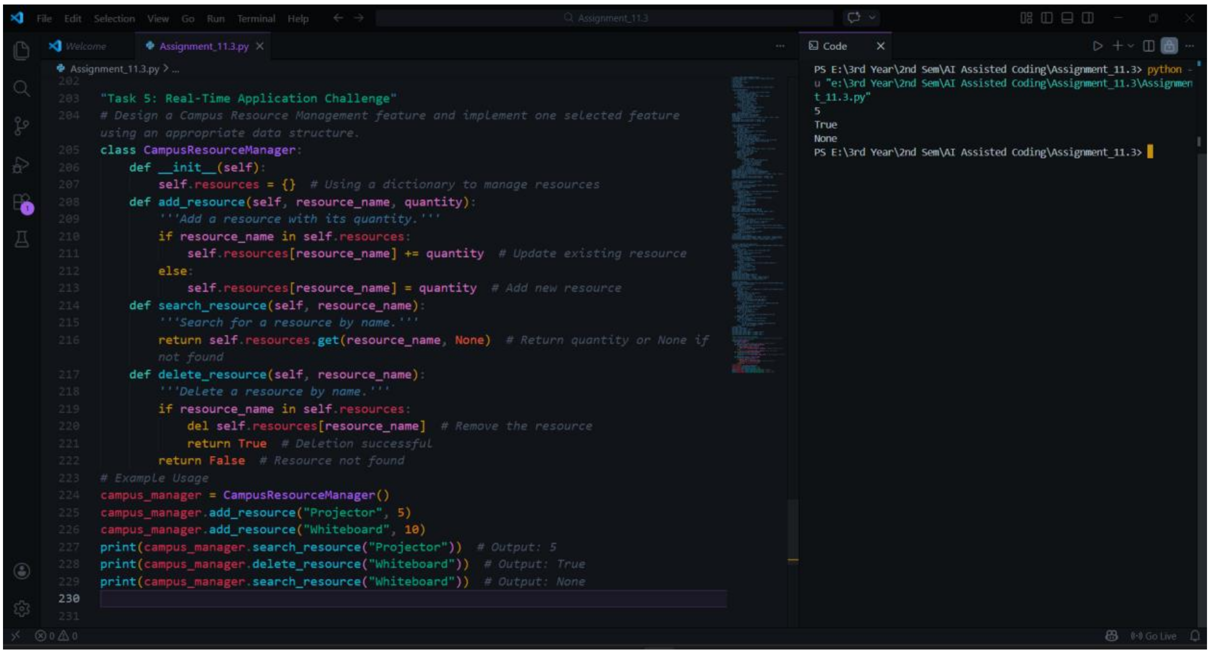
The hash table stores data using a hashing function to determine storage index. Collision handling is done using chaining, allowing multiple elements per bucket. This ensures efficient average-time operations. AI helped generate structured bucket management logic.

Task 5: Real-Time Application Challenge

Prompt:

Design a Campus Resource Management feature and implement one selected feature using an appropriate data structure.

Code & Output:



```python

"Task 5: Real-Time Application Challenge"
# Design a Campus Resource Management feature and implement one selected feature using an appropriate data structure.
class CampusResourceManager:
    def __init__(self):
        self.resources = {}  # Using a dictionary to manage resources
    def add_resource(self, resource_name, quantity):
        '''Add a resource with its quantity.'''
        if resource_name in self.resources:
            self.resources[resource_name] += quantity  # Update existing resource
        else:
            self.resources[resource_name] = quantity  # Add new resource
    def search_resource(self, resource_name):
        '''Search for a resource by name.'''
        return self.resources.get(resource_name, None)  # Return quantity or None if not found
    def delete_resource(self, resource_name):
        '''Delete a resource by name.'''
        if resource_name in self.resources:
            del self.resources[resource_name]  # Remove the resource
            return True  # Deletion successful
        return False  # Resource not found
# Example Usage
campus_manager = CampusResourceManager()
campus_manager.add_resource("Projector", 5)
campus_manager.add_resource("Whiteboard", 10)
print(campus_manager.search_resource("Projector"))  # Output: 5
print(campus_manager.delete_resource("Whiteboard"))  # Output: True
print(campus_manager.search_resource("Whiteboard"))  # Output: None


```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3\Assignment_11.3.py"
5
True
None
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_11.3>
```

Explanation:

The cafeteria system uses a queue to maintain FIFO order of service. Customers are served in the order they arrive, ensuring fairness. This data structure matches real-world queue behavior. AI assistance helped implement and structure the queue methods efficiently.

Final Conclusion:

This lab demonstrated implementation of fundamental data structures using AI assistance. Structures such as arrays, linked lists, stacks, queues, priority queues, and hash tables were explored in practical scenarios. AI tools improved code clarity and development speed. However, logical understanding and correct structure selection remain essential responsibilities of the developer.