

# Assignment 11.3

Name : Karthikeya uthuri

Ht.no: 2303A51306

Batch: 05

## Task 1:

Smart Contact Manager (Arrays & Linked Lists)

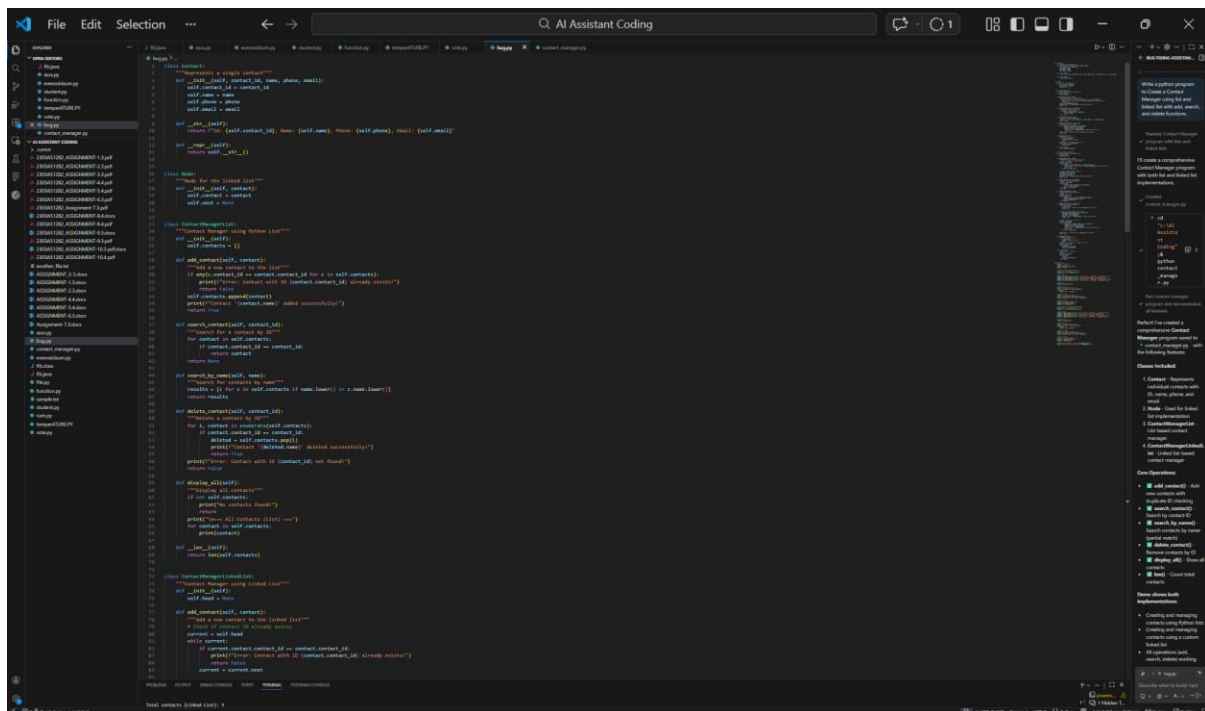
Scenario

SR University's student club requires a simple Contact Manager Application to store members' names and phone numbers. The system should support efficient addition, searching, and deletion of contacts.

## Prompt:

Write a python program to Create a Contact Manager using list and linked list with add, search, and delete functions.

## Code:



```
class Contact:
    """Represent a single contact"""
    def __init__(self, contact_id, name, phone, email):
        self.contact_id = contact_id
        self.name = name
        self.phone = phone
        self.email = email

    def __str__(self):
        return f'({self.contact_id}, {self.name}, {self.phone}, {self.email})'

    def __repr__(self):
        return self.__str__()

class ContactManager:
    """Contact Manager using Python List"""
    def __init__(self):
        self.contacts = []

    def add_contact(self, contact):
        """Add a new contact to the list"""
        if not contact_id or not name or not phone or not email:
            print("Error: Contact with ID, Name, Phone, Email already exists")
            return False
        self.contacts.append(contact)
        print(f"Contact {contact.name} added successfully")
        return True

    def search_contact(self, contact_id):
        """Search for a contact by ID"""
        for contact in self.contacts:
            if contact.contact_id == contact_id:
                return contact
        return None

    def search_by_name(self, name):
        """Search for contacts by name"""
        results = []
        for c in self.contacts:
            if name.lower() in c.name.lower():
                results.append(c)
        return results

    def delete_contact(self, contact_id):
        """Delete a contact by ID"""
        for i, contact in enumerate(self.contacts):
            if contact.contact_id == contact_id:
                del self.contacts[i]
                print(f"Contact {contact.name} deleted successfully")
                return True
        print("Error: Contact with ID {contact_id} not found")
        return False

    def display_all(self):
        """Display all contacts"""
        if not self.contacts:
            print("No contacts found")
        else:
            print("All contacts in list:")
            for contact in self.contacts:
                print(contact)

    def __str__(self):
        return str(self.contacts)

    def __repr__(self):
        return self.__str__()

# Contact Manager using Python List
cm = ContactManager()

# Add contacts
cm.add_contact(Contact(1, "John", 1234567890, "john@example.com"))
cm.add_contact(Contact(2, "Jane", 9876543210, "jane@example.com"))
cm.add_contact(Contact(3, "Bob", 5555555555, "bob@example.com"))

# Search contacts
cm.search_contact(1)
cm.search_contact(2)
cm.search_by_name("John")

# Delete contact
cm.delete_contact(1)

# Display all contacts
cm.display_all()
```

```
File Edit Selection *** Q AI Assistant Coding
// ContactManager.h
#pragma once
#include <string>
#include <vector>
using namespace std;

// ContactManager class
class ContactManager {
public:
    // Constructor
    ContactManager() {}

    // Add a new contact
    void addContact(const string& name, const string& phone, const string& email) {
        // Add logic here
    }

    // Search for a contact by name
    vector<Contact> searchBy(const string& name) const {
        // Search logic here
    }

    // Delete a contact
    void deleteContact(const string& name) {
        // Delete logic here
    }

    // Display all contacts
    void displayAll() const {
        // Display logic here
    }

    // Get the total number of contacts
    int getTotalContacts() const {
        // Get total contacts logic here
    }
};

// Contact struct
struct Contact {
    string name;
    string phone;
    string email;
};

// Main function
int main() {
    ContactManager manager;

    // Add some contacts
    manager.addContact("John Doe", "123-456-7890", "john.doe@example.com");
    manager.addContact("Jane Smith", "987-654-3210", "jane.smith@example.com");
    manager.addContact("Bob Johnson", "555-555-5555", "bob.johnson@example.com");

    // Search for a contact
    vector<Contact> results = manager.searchBy("John");
    if (!results.empty()) {
        for (const auto& contact : results) {
            cout << contact.name << " | " << contact.phone << " | " << contact.email << endl;
        }
    }

    // Display all contacts
    manager.displayAll();

    // Get total contacts
    int total = manager.getTotalContacts();
    cout << "Total contacts: " << total << endl;

    return 0;
}
```

```
File Edit Selection *** Q AI Assistant Coding
// ContactManager.cpp
#include "ContactManager.h"
#include <iostream>
using namespace std;

// ContactManager implementation
ContactManager::ContactManager() {}

// Add a new contact
void ContactManager::addContact(const string& name, const string& phone, const string& email) {
    Contact contact;
    contact.name = name;
    contact.phone = phone;
    contact.email = email;
    contacts.push_back(contact);
}

// Search for a contact by name
vector<Contact> ContactManager::searchBy(const string& name) const {
    vector<Contact> results;
    for (const auto& contact : contacts) {
        if (contact.name == name) {
            results.push_back(contact);
        }
    }
    return results;
}

// Delete a contact
void ContactManager::deleteContact(const string& name) {
    for (auto it = contacts.begin(); it != contacts.end(); ++it) {
        if (*it).name == name) {
            contacts.erase(it);
            break;
        }
    }
}

// Display all contacts
void ContactManager::displayAll() const {
    for (const auto& contact : contacts) {
        cout << contact.name << " | " << contact.phone << " | " << contact.email << endl;
    }
}

// Get the total number of contacts
int ContactManager::getTotalContacts() const {
    return contacts.size();
}
```

Output:



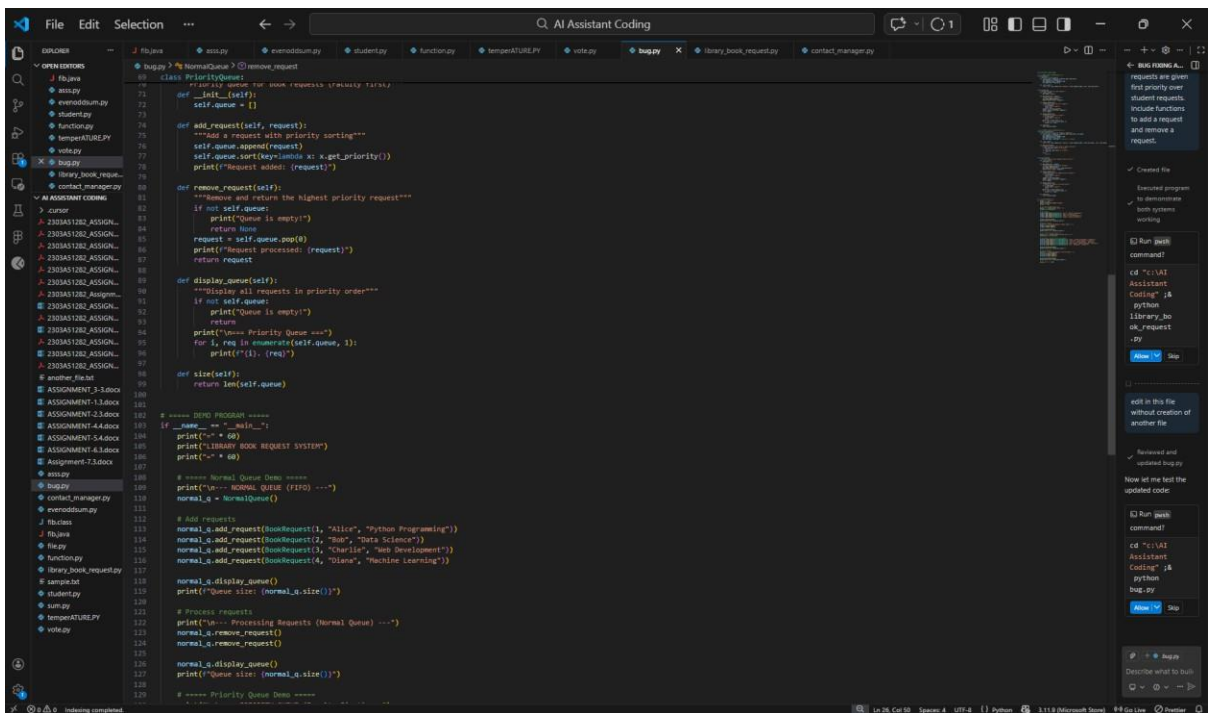
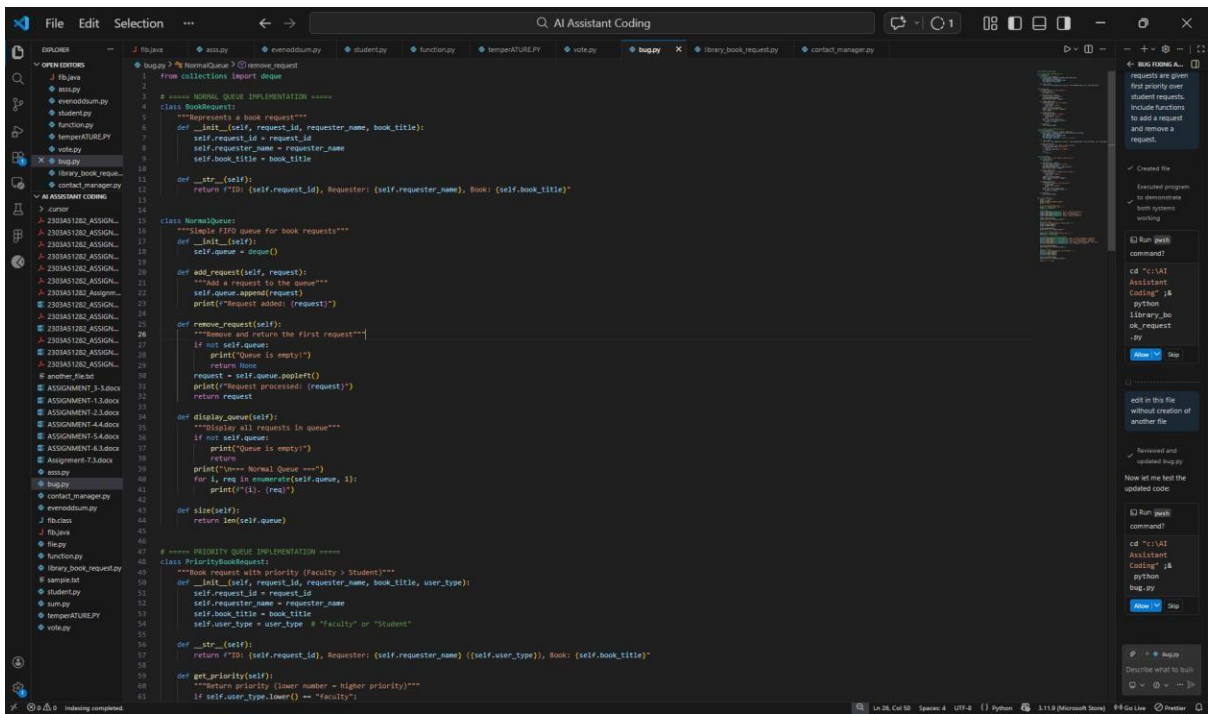
- ## Task 2:

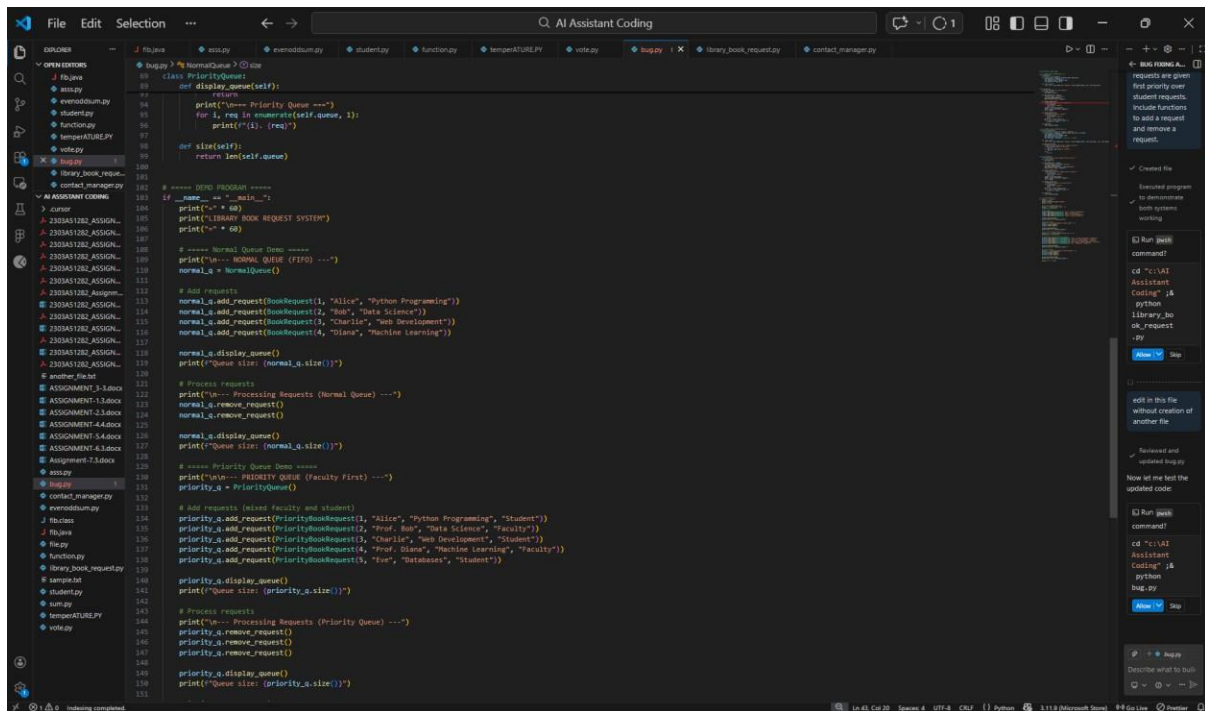
## Scenario

The SRU Library manages book borrow requests. Students and faculty submit requests, but faculty requests must be prioritized over student requests.

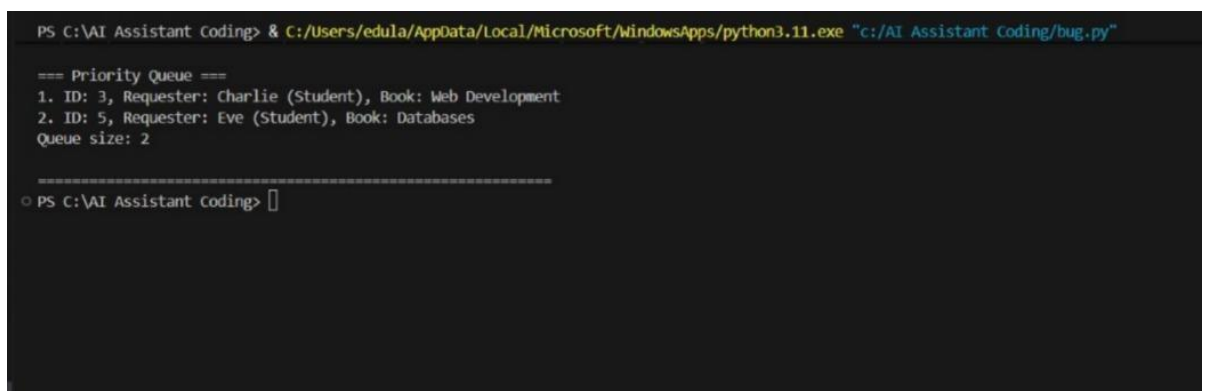
Write a Python program for a library book request system. First, make a normal queue where requests are handled in the order they come. Then, make another version where faculty requests are given first priority over student requests. Include functions to add a request and remove a request.

**Code:**





## Output:



## Explanation:

- Queue (FIFO) → First request comes, first served.(If a student requests first, they get the book first.)
- Priority Queue → Faculty requests are served before students, even if they come later.
- enqueue() → Adds a request to the system.
- dequeue() → Removes and processes the next request.

## Task 3: Emergency Help Desk (Stack Implementation)

### Scenario

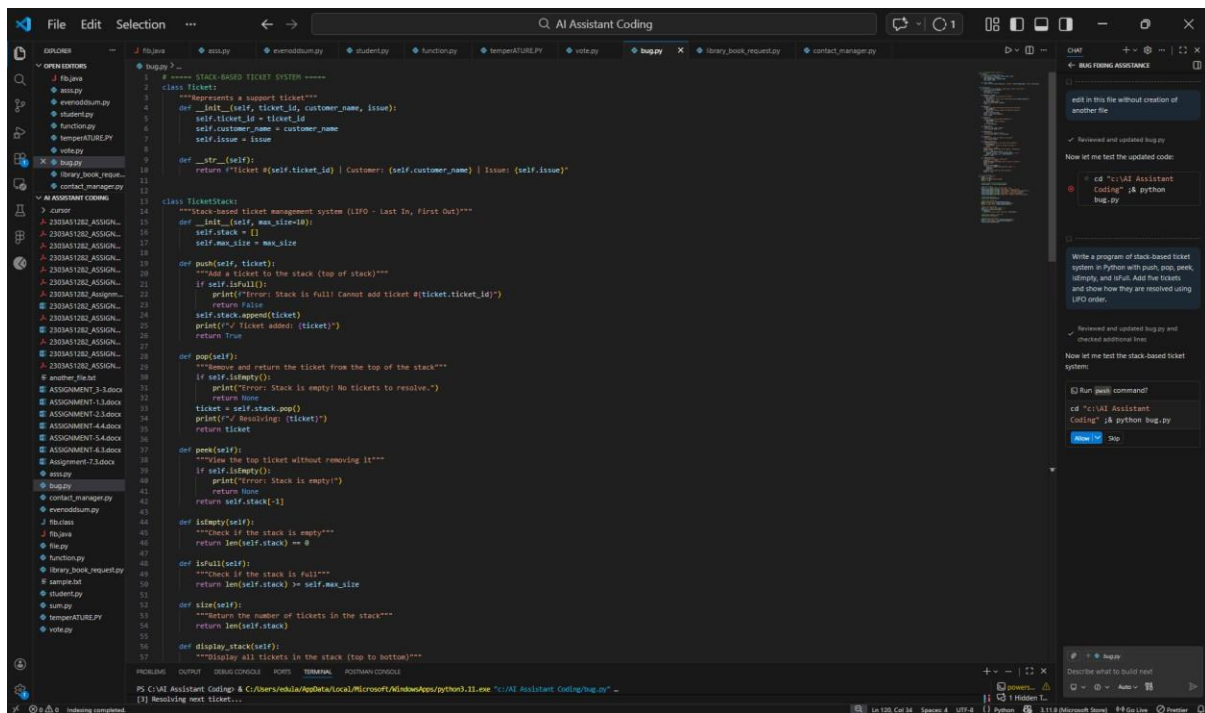
SR University's IT Help Desk receives technical support tickets from students and staff.

While tickets are received sequentially, issue escalation follows a Last-In, First-Out (LIFO) approach.

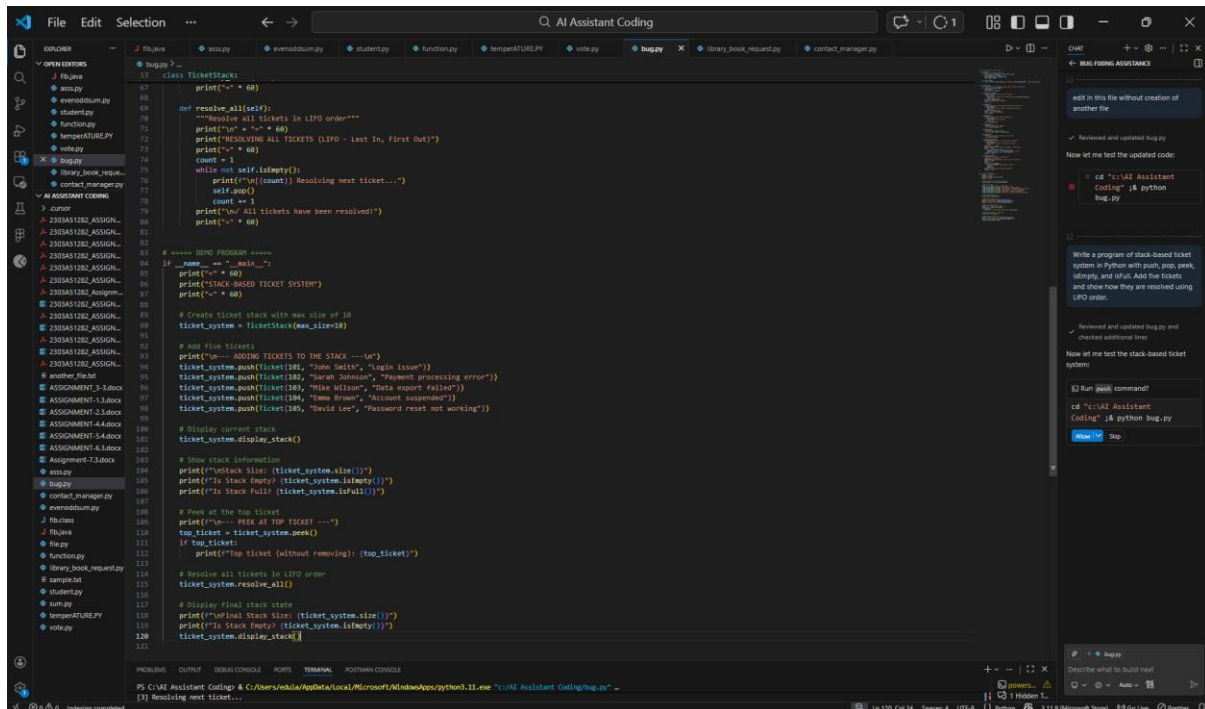
## Prompt:

Write a program of stack-based ticket system in Python with push, pop, peek, isEmpty, and isFull. Add five tickets and show how they are resolved using LIFO order.

## Code:

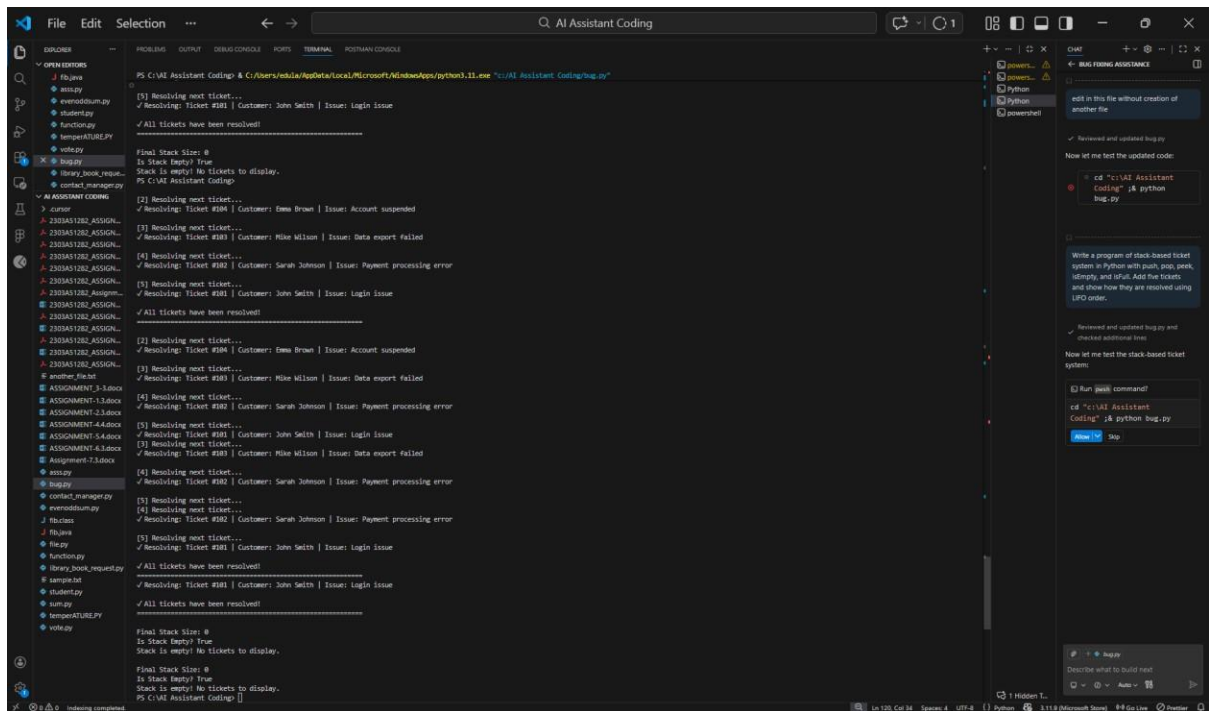


```
1 # ===== STACK-BASED TICKET SYSTEM =====
2
3 """Represents a support ticket"""
4 class Ticket:
5     def __init__(self, ticket_id, customer_name, issue):
6         self.ticket_id = ticket_id
7         self.customer_name = customer_name
8         self.issue = issue
9
10     def __str__(self):
11         return f"Ticket #{self.ticket_id} | Customer: {self.customer_name} | Issue: {self.issue}"
12
13
14 class TicketStack:
15     """Stack-based ticket management system (LIFO - Last In, First Out)"""
16     def __init__(self, max_size=10):
17         self.stack = []
18         self.max_size = max_size
19
20     def push(self, ticket):
21         """Add a ticket to the stack (top of stack)"""
22         if self.is_full():
23             print(f"Error: Stack is full! Cannot add ticket #{ticket.ticket_id}")
24             return False
25         self.stack.append(ticket)
26         print(f"/\ Ticket added: {ticket}")
27         return True
28
29     def pop(self):
30         """Remove and return the ticket from the top of the stack"""
31         if self.is_empty():
32             print(f"Error: Stack is empty! No tickets to resolve.")
33             return None
34         ticket = self.stack.pop()
35         print(f"/\ Resolving: {ticket}")
36         return ticket
37
38     def peek(self):
39         """View the top ticket without removing it"""
40         if self.is_empty():
41             print(f"Error: Stack is empty!")
42             return None
43         return self.stack[-1]
44
45     def is_empty(self):
46         """Check if the stack is empty"""
47         return len(self.stack) == 0
48
49     def is_full(self):
50         """Check if the stack is full"""
51         return len(self.stack) >= self.max_size
52
53     def size(self):
54         """Return the number of tickets in the stack"""
55         return len(self.stack)
56
57     def display_stack(self):
58         """Display all tickets in the stack (top to bottom)"""
```



```
59
60
61     def resolve_all(self):
62         """Resolve all tickets in LIFO order"""
63         print(f"/\n * ~ ~ ~ *")
64         print(f"Resolving ALL TICKETS (LIFO - Last In, First Out)")
65         count = 1
66         while not self.is_empty():
67             print(f"/\n {count} Resolving next ticket...")
68             self.pop()
69             count += 1
70         print(f"/\n All tickets have been resolved!")
71         print(f"/\n * ~ ~ ~ *")
72
73
74 # ===== TEST PROGRAM =====
75 if __name__ == "__main__":
76     print(f"/\n * ~ ~ ~ *")
77     print(f"STACK-BASED TICKET SYSTEM")
78     print(f"/\n * ~ ~ ~ *")
79
80     # Create ticket stack with max size of 10
81     ticket_system = TicketStack(max_size=10)
82
83     # Add five tickets
84     print(f"/\n --- ADDING TICKETS TO THE STACK ---")
85     ticket_system.push(ticket_id=101, customer_name="John Smith", issue="Login issue")
86     ticket_system.push(ticket_id=102, customer_name="Sarah Johnson", issue="Payment processing error")
87     ticket_system.push(ticket_id=103, customer_name="Mike Wilson", issue="Data export failed")
88     ticket_system.push(ticket_id=104, customer_name="Emma Brown", issue="Account suspension")
89     ticket_system.push(ticket_id=105, customer_name="David Lee", issue="Password reset not working")
90
91     # Display current stack
92     ticket_system.display_stack()
93
94     # Show stack information
95     print(f"/\n Stack Size: {ticket_system.size()}")
96     print(f"/\n Is Stack Empty? {ticket_system.is_empty()}")
97     print(f"/\n Is Stack Full? {ticket_system.is_full()}")
98
99     # Peek at the top ticket
100     print(f"/\n --- PEAK AT TOP TICKET ---")
101     top_ticket = ticket_system.peek()
102     if top_ticket:
103         print(f"/\n Top ticket (without removing): {top_ticket}")
104
105     # Resolve all tickets in LIFO order
106     ticket_system.resolve_all()
107
108     # Display final stack state
109     print(f"/\n Final Stack Size: {ticket_system.size()}")
110     print(f"/\n Is Stack Empty? {ticket_system.is_empty()}")
111     ticket_system.display_stack()
112
113
114 # ===== END OF PROGRAM =====
```

## Output:



## Explanation:

The program uses a stack to manage help desk tickets.

A stack works in last in, first solved order.

When a new ticket is raised, it is added to the top.

When solving a ticket, the most recent one is handled first.

The program can also check if there are no tickets left or if the stack is full.

## Task 4:

Hash Table

### Objective

To implement a Hash Table and understand collision handling.

### Prompt:

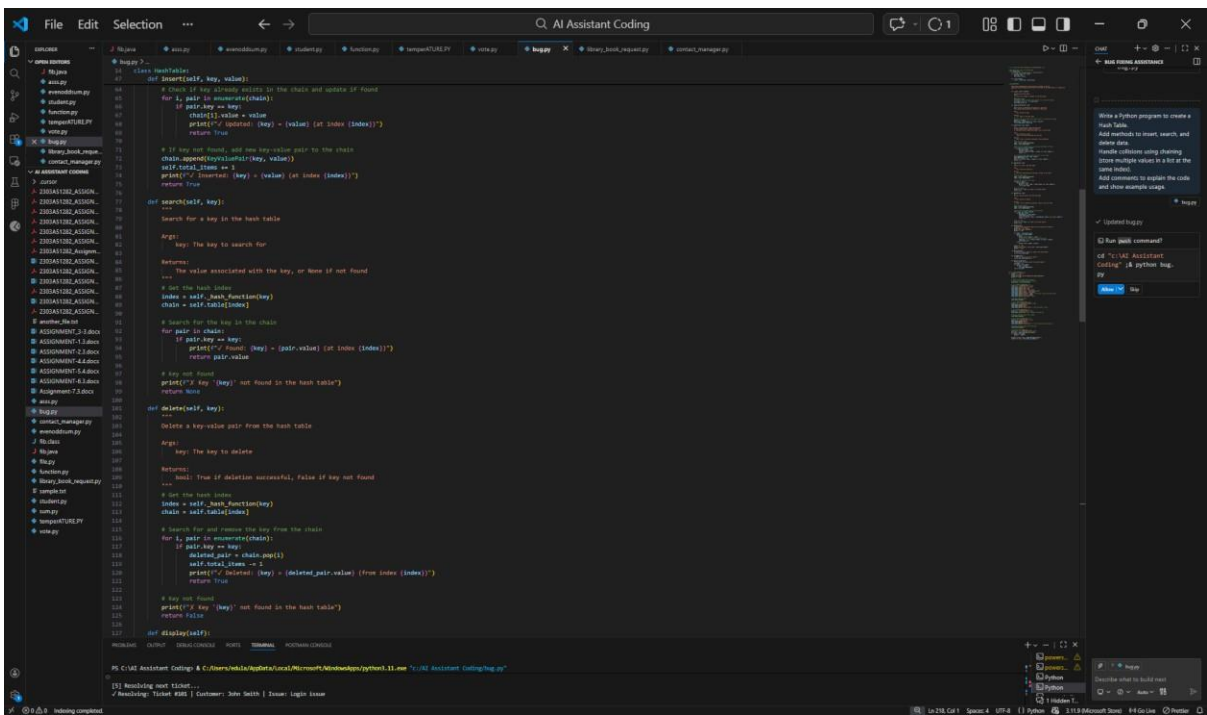
Write a Python program to create a Hash Table.

Add methods to insert, search, and delete data.

Handle collisions using chaining (store multiple values in a list at the same index).

Add comments to explain the code and show example usage.

### Code:



```
def get_all_items(self):  
    """Returns all key-value pairs in the hash table"""  
    all_items = []  
    for chain in self.table:  
        for pair in chain:  
            all_items.append(pair)  
    return all_items  
  
# ===== DYNAMIC PROGRAM =====  
if __name__ == "__main__":  
    print("\n=====")  
    print("\n=====")  
    print("\n=====")  
    # Create a hash table with 5 buckets  
    # (will also be demonstrating collisions)  
    hash_table = HashTable(5)  
    # ===== INSERT OPERATIONS =====  
    print("\n--- INSERTING DATA ---")  
    hash_table.insert("name", "Alice")  
    hash_table.insert("age", 30)  
    hash_table.insert("city", "New York")  
    hash_table.insert("email", "alice@email.com")  
    hash_table.insert("phone", "555-1234")  
    hash_table.insert("country", "USA") # This may collide with other keys  
    hash_table.insert("salary", 7000)  
    hash_table.insert("department", "IT")  
    # Display the hash table  
    hash_table.display()  
    # ===== SEARCH OPERATIONS =====  
    print("\n--- SEARCHING FOR DATA ---")  
    hash_table.search("name")  
    hash_table.search("age")  
    hash_table.search("unknown_key") # Key that doesn't exist  
    # ===== UPDATE OPERATIONS =====  
    print("\n--- UPDATING DATA ---")  
    hash_table.update("age", 31) # Update existing key  
    # Display the hash table after update  
    hash_table.display()  
    # ===== DELETE OPERATIONS =====  
    print("\n--- DELETING DATA ---")  
    hash_table.delete("email")  
    hash_table.delete("city")  
    hash_table.delete("nonexistent") # Try to delete non-existent key  
    # Display the hash table after deletions  
    hash_table.display()  
    # ===== GET ALL ITEMS =====  
    print("\n--- ALL REMAINING ITEMS ---")  
    all_items = hash_table.get_all_items()  
    for item in all_items:  
        print(item)  
    print("\nInitial items: ", hash_table.get_size())  
    print("\nIs empty: ", hash_table.is_empty())
```

## Output:

```
PS C:\AI Assistant Codings> C:\Users\adiba\AppData\Local\Microsoft\WindowsApps\python11.exe "C:\AI Assistant Codings\bug.py"  
Index 0: [empty]  
Index 1: age: 31 -> city: New York -> department: IT  
Index 2: name: Alice -> salary: 7000  
Index 3: phone: 555-1234 -> country: USA  
Index 4: [empty]  
Total items in hash table: 6  
-----  
--- SEARCHING FOR DATA ---  
✓ Found: name = Alice (at index 2)  
✓ Found: age = 31 (at index 1)  
✗ Key 'unknown_key' not found in the hash table  
-----  
--- UPDATING DATA ---  
✓ Updated: age = 31 (at index 1)  
-----  
HASH TABLE CONTENTS  
Index 0: email: alice@email.com  
Index 1: age: 31 -> city: New York -> department: IT  
Index 2: name: Alice -> salary: 7000  
Index 3: phone: 555-1234 -> country: USA  
Index 4: [empty]  
Total items in hash table: 6  
-----  
--- DELETING DATA ---  
✓ Deleted: email = alice@email.com (from index 0)  
✓ Deleted: city = New York (from index 1)  
✗ Key 'nonexistent' not found in the hash table  
-----  
HASH TABLE CONTENTS  
Index 0: [empty]  
Index 1: age: 31  
Index 2: name: Alice -> department: IT  
Index 3: phone: 555-1234 -> country: USA  
Index 4: [empty]  
Total items in hash table: 4  
-----  
--- ALL REMAINING ITEMS ---  
age: 31  
department: IT  
name: Alice  
salary: 7000  
phone: 555-1234  
country: USA  
Total items: 6  
Is empty: False  
PS C:\AI Assistant Codings>
```

## Explanation:

- A Hash Table stores data using a key and value.
- A hash function decides where to store the data.
- Sometimes two keys go to the same place. This is called a collision.
- To solve collisions, we use chaining, meaning we store multiple items in a list at the same index.
- The program should allow adding, finding, and removing data correctly.

## Task 5:

### Real-Time Application Challenge Scenario

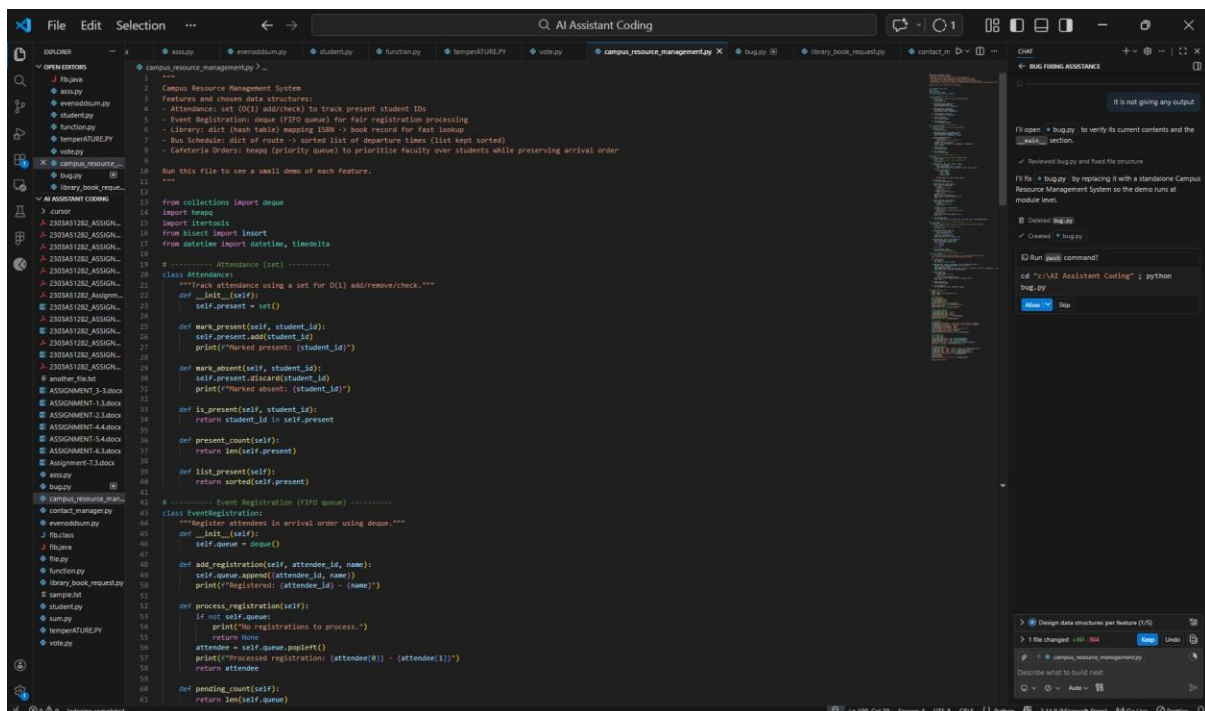
Design a Campus Resource Management System with the following features:

- Student Attendance Tracking
- Event Registration System
- Library Book Borrowing
- Bus Scheduling System
- Cafeteria Order Queue

### Prompt:

Create a Campus Resource Management System in Python. For each feature (Attendance, Event Registration, Library, Bus Schedule, Cafeteria Orders), choose the best data structure

### Code:



```
File Edit Selection ... Q AI Assistant Coding
campus_resource_management.py ...
Campus Resource Management System
Features and chosen data structures:
- Attendance: set (O(1) add/check) to track present student IDs
- Event Registration: deque (FIFO queue) for fair registration processing
- Library: dict (hash table) mapping ISBN -> book record for fast lookup
- Bus Schedules: dict of route -> sorted list of departure times (list kept sorted)
- Cafeteria Orders: heap (priority queue) to prioritize faculty over students while preserving arrival order

Run this file to see a small demo of each feature.
...
from collections import deque
import heapq
import heapq
import heapq
from heapq import heapify
from heapq import heapreplace
from heapq import heappop
from heapq import heappush
from heapq import heappushpop
from heapq import heapreplace

# Attendance (set)
class Attendance:
    """Track attendance using a set for O(1) add/remove/check."""
    def __init__(self):
        self.present = set()

    def mark_present(self, student_id):
        self.present.add(student_id)
        print(f"Marked present: {student_id}")

    def mark_absent(self, student_id):
        self.present.discard(student_id)
        print(f"Marked absent: {student_id}")

    def is_present(self, student_id):
        return student_id in self.present

    def present_count(self):
        return len(self.present)

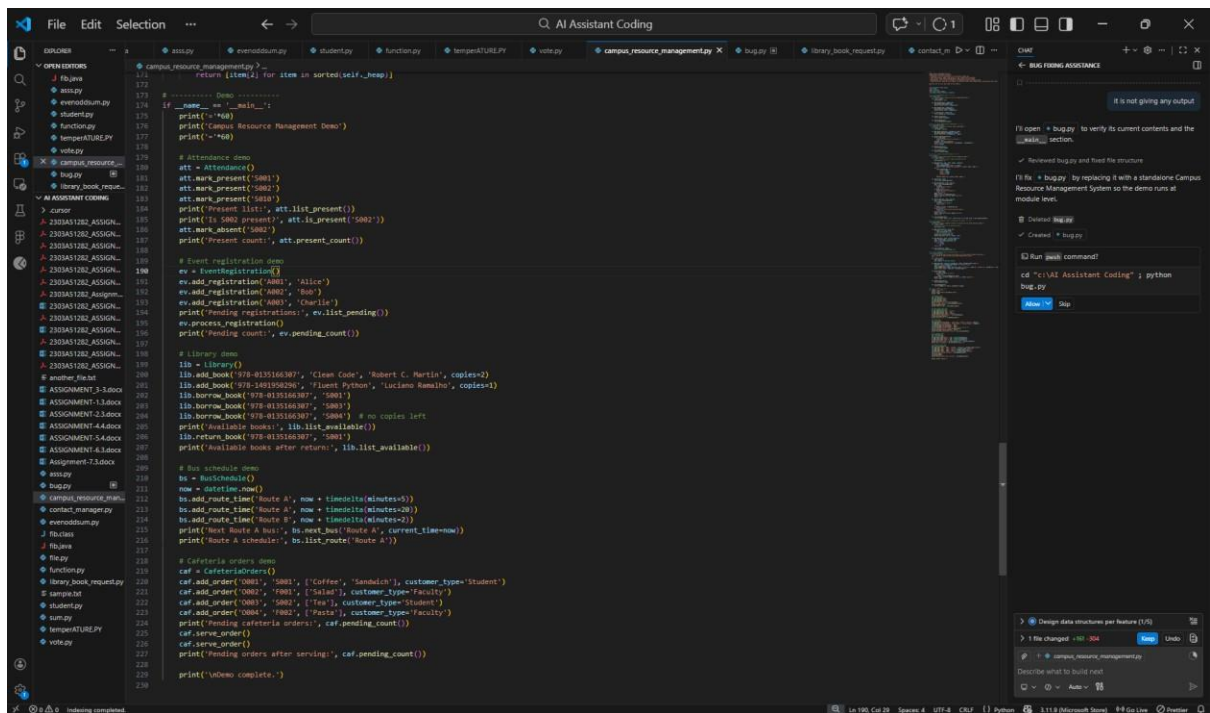
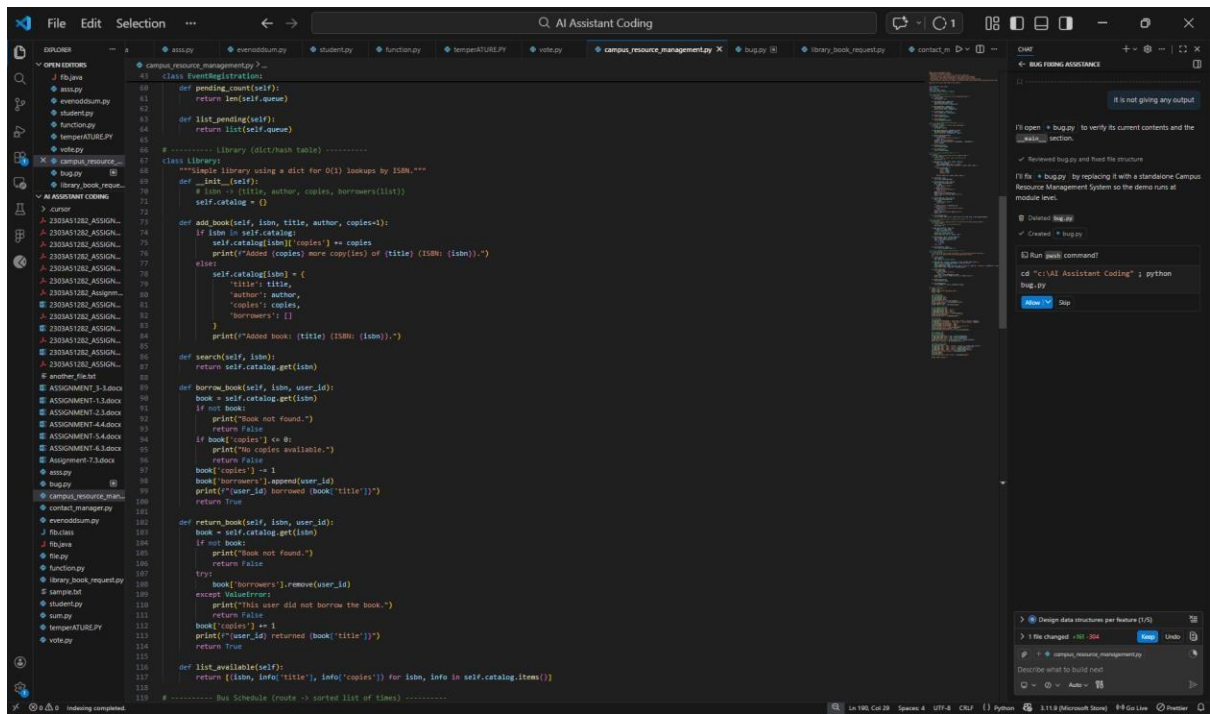
    def list_present(self):
        return sorted(self.present)

# Event Registration (FIFO queue)
class EventRegistration:
    """Register attendees in arrival order using deque."""
    def __init__(self):
        self.queue = deque()

    def add_registration(self, attendee_id, name):
        self.queue.append((attendee_id, name))
        print(f"Registered: {attendee_id} - {name}")

    def process_registration(self):
        if not self.queue:
            print("No registrations to process.")
            return None
        attendee = self.queue.popleft()
        print(f"Processed registration: {attendee[0]} - {attendee[1]}")
        return attendee

    def pending_count(self):
        return len(self.queue)
```



Output:

```
PROBLEMS OUTPUT DEBUG-CONSOLE PORTS TERMINAL POSTMAN CONSOLE

PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bug.py"
Is empty: False
PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bug.py"
PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/campus_resource_management.py"

=====
Campus Resource Management Demo
=====

Marked present: 5001
Marked present: 5002
Marked present: 5010
Present list: ['5001', '5002', '5010']
Is 5002 present? True
Marked absent: 5002
Present count: 2
Registered: A001 - Alice
Registered: A002 - Bob
Registered: A003 - Charlie
Pending registrations: [('A001', 'Alice'), ('A002', 'Bob'), ('A003', 'Charlie')]
Processed registration: A001 - Alice
Pending count: 2
Added book: Clean Code (ISBN: 978-0135166307).
Added book: Fluent Python (ISBN: 978-1491958296).
5001 borrowed Clean Code
5003 borrowed Clean Code
No copies available.
Available books: [('978-0135166307', 'Clean Code', 0), ('978-1491958296', 'Fluent Python', 1)]
5001 returned Clean Code
Available books after return: [('978-0135166307', 'Clean Code', 1), ('978-1491958296', 'Fluent Python', 1)]
Added bus time for Route A: 2026-02-18 10:42:24.367227
Added bus time for Route A: 2026-02-18 10:57:24.367227
Added bus time for Route B: 2026-02-18 10:39:24.367227
Next Route A bus: 2026-02-18 10:42:24.367227
Route A schedule: [datetime.datetime(2026, 2, 18, 10, 42, 24, 367227), datetime.datetime(2026, 2, 18, 10, 57, 24, 367227)]
Order added: 0001 (Student)
Order added: 0002 (Faculty)
Order added: 0003 (Student)
Order added: 0004 (Faculty)
Pending cafeteria orders: 4
Serving order: 0002 (Faculty)
Serving order: 0004 (Faculty)
Pending orders after serving: 2

Demo complete.
PS C:\AI Assistant Coding> 
```

## Explanation:

Library Book Borrowing using a queue:

- The queue stores student names who request a book.
- When a student requests a book, we use `enqueue()` to add them to the queue.
- When a book becomes available, we use `dequeue()` to give it to the first student in line.
- This ensures fairness because the first requester gets the book first.

