

Assignment 11.3 Ai Assisted Coding

Ht.no: 2303A510C0

Batch: 06

Task 1:

Smart Contact Manager (Arrays & Linked Lists)

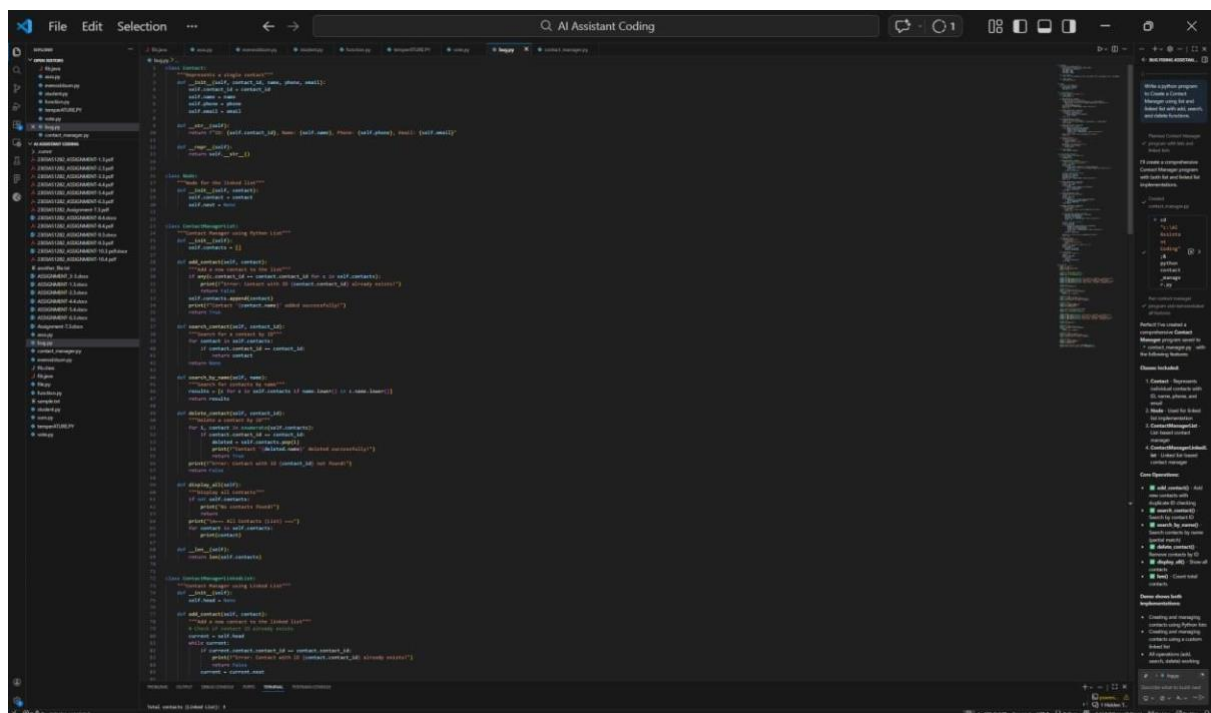
Scenario

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.

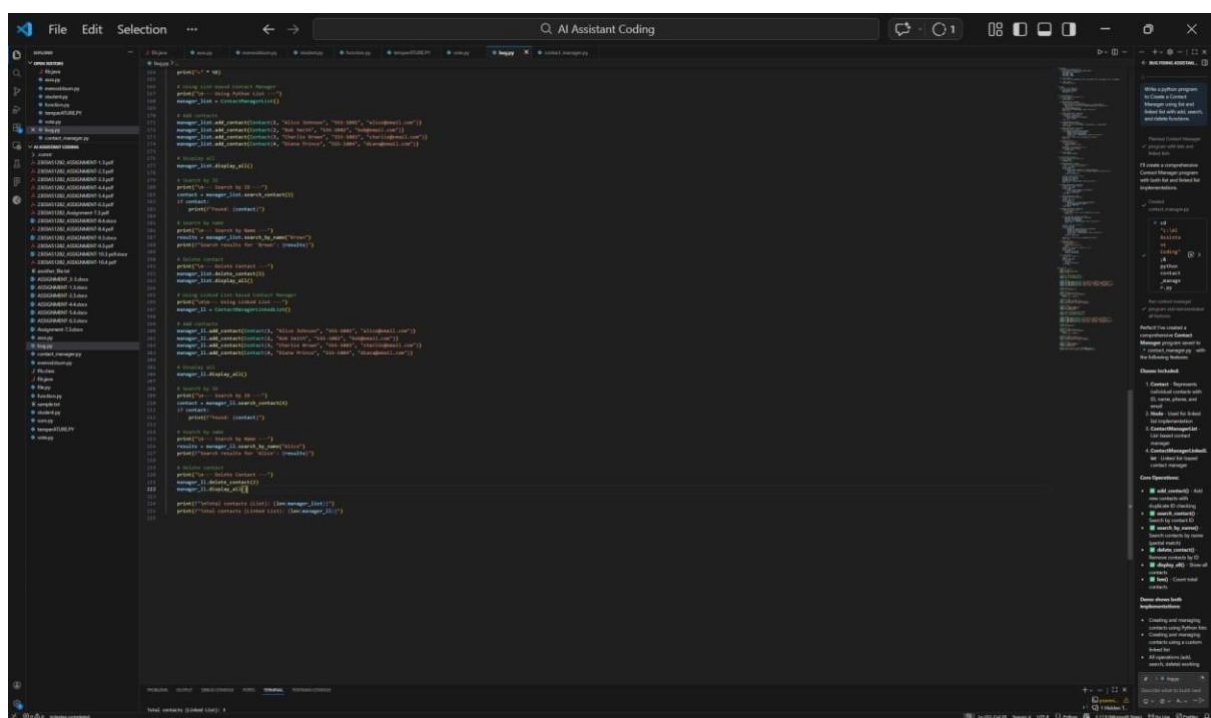
Prompt:

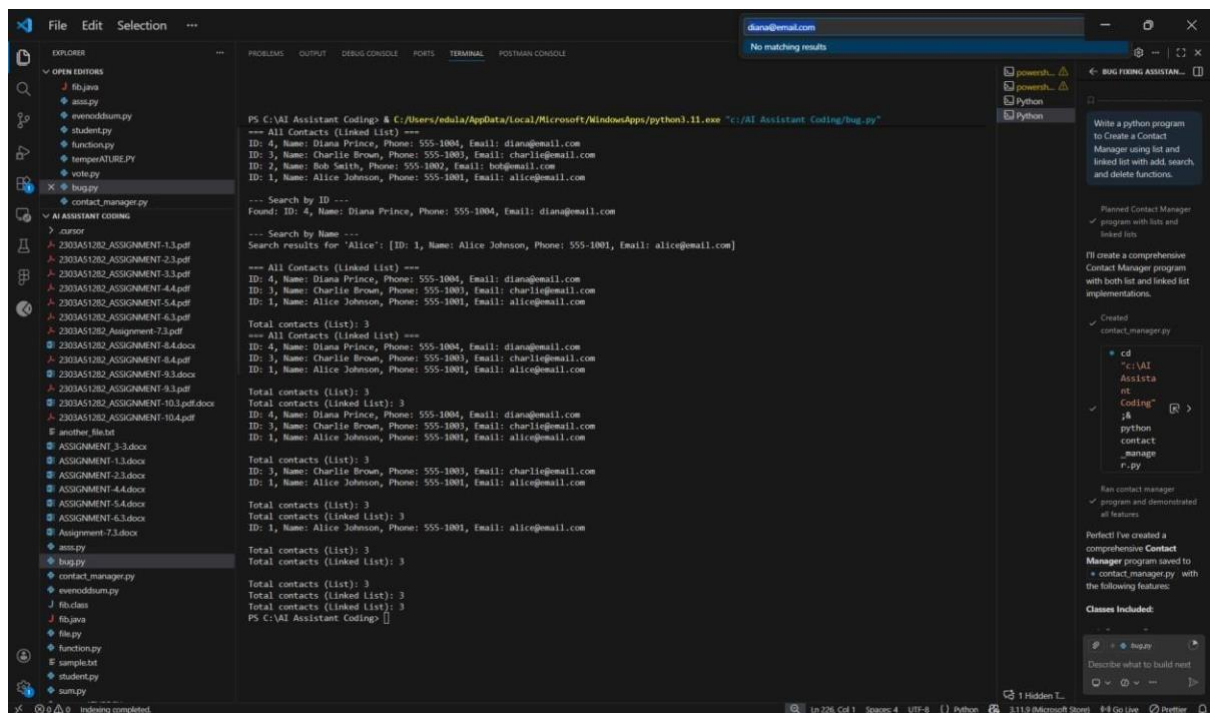
Write a python program to Create a Contact Manager using list and linked list with add, search, and delete functions.

Code:



```
1 class ContactManager:
2     def __init__(self):
3         self.head = None
4         self.tail = None
5         self.size = 0
6
7     def add_contact(self, name, phone, email):
8         """Add a new contact to the linked list"""
9         new_contact = Contact(name, phone, email)
10        if self.head is None:
11            self.head = new_contact
12            self.tail = new_contact
13        else:
14            self.tail.next = new_contact
15            self.tail = new_contact
16        self.size += 1
17
18    def search_contact(self, name):
19        """Search for a contact by name"""
20        current = self.head
21        while current is not None:
22            if current.name == name:
23                return current
24            current = current.next
25        return None
26
27    def delete_contact(self, name):
28        """Delete a contact by name"""
29        if self.head is None:
30            return
31        if self.head.name == name:
32            self.head = self.head.next
33        else:
34            current = self.head
35            while current.next is not None:
36                if current.next.name == name:
37                    current.next = current.next.next
38                current = current.next
39        self.size -= 1
40
41    def display_all_contacts(self):
42        """Display all contacts in the linked list"""
43        current = self.head
44        while current is not None:
45            print(f"Name: {current.name}, Phone: {current.phone}, Email: {current.email}")
46            current = current.next
47
48    def __str__(self):
49        return f"Contact Manager with {self.size} contacts"
50
51 if __name__ == "__main__":
52     manager = ContactManager()
53     manager.add_contact("John Doe", "1234567890", "john.doe@example.com")
54     manager.add_contact("Jane Smith", "9876543210", "jane.smith@example.com")
55     manager.add_contact("Bob Johnson", "5555555555", "bob.johnson@example.com")
56     manager.display_all_contacts()
57     manager.search_contact("John Doe")
58     manager.delete_contact("John Doe")
59     manager.display_all_contacts()
60     print(manager)
```





Explanation:

- In an array, adding at the end is fast, but inserting in the middle is slow because elements must shift.
- In a linked list, insertion is fast because no shifting is needed.
- Searching takes the same time in both (you must check each element).
- Deleting in an array is slower due to shifting elements.
- Linked list is better for frequent insertions and deletions.

Task 2:

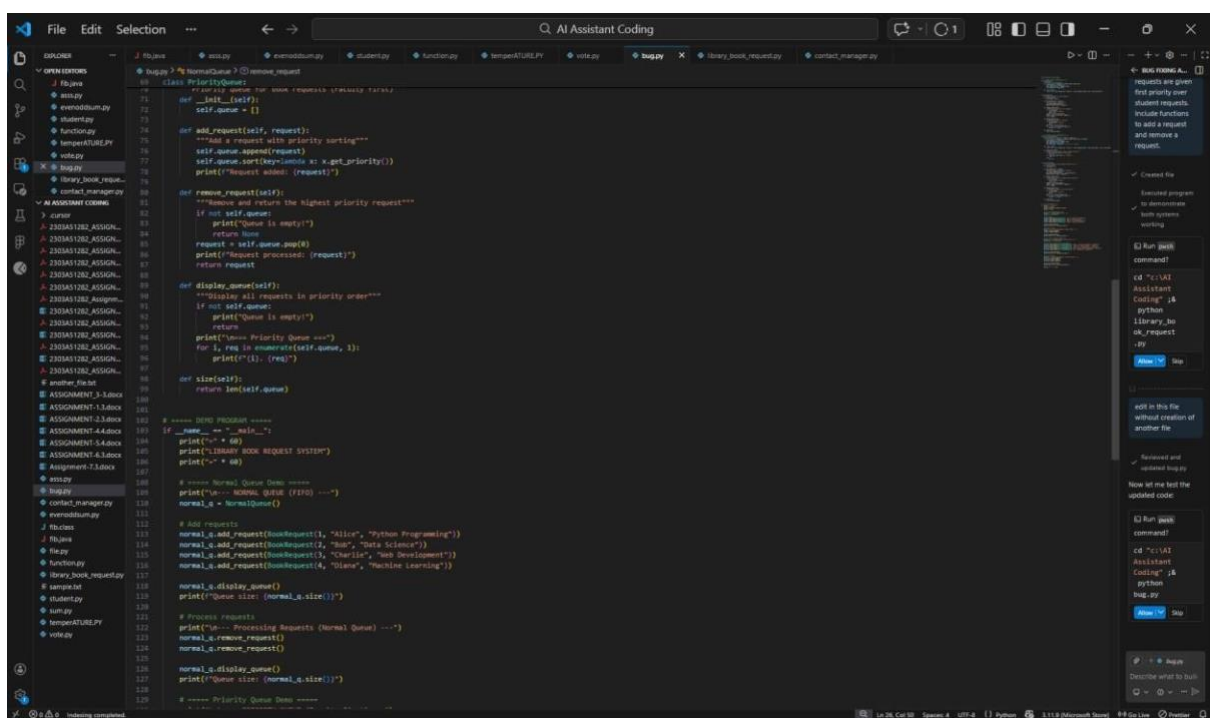
Library Book Search System (Queues & Priority Queues) Scenario

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

Prompt:

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:



```

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

    def enqueue(self, request):
        self.queue.append(request)

    def dequeue(self):
        if self.queue:
            return self.queue.pop(0)
        return None

    def display(self):
        print("Queue size: ", len(self.queue))
        for i, req in enumerate(self.queue, 1):
            print(f"{i}. ID: {req['id']}, Requester: {req['requester']}, Book: {req['book']}")

# ===== PRIORITY QUEUE =====
if __name__ == "__main__":
    p_queue = PriorityQueue()

    # Add requests
    p_queue.enqueue({"id": 1, "requester": "Alice", "book": "Python Programming"})
    p_queue.enqueue({"id": 2, "requester": "Bob", "book": "Data Science"})
    p_queue.enqueue({"id": 3, "requester": "Charlie", "book": "Web Development"})
    p_queue.enqueue({"id": 4, "requester": "David", "book": "Machine Learning"})

    # Display queue
    p_queue.display()

    # Process requests
    print("Processing Requests (Priority Queue) ---")
    while True:
        req = p_queue.dequeue()
        if req:
            print(f"Request processed: {req}")
        else:
            break

    # Add requests (Faculty and Student)
    p_queue.enqueue({"id": 5, "requester": "Prof. Smith", "book": "Database Systems", "priority": "Faculty"})
    p_queue.enqueue({"id": 6, "requester": "Alice", "book": "Python Programming", "priority": "Student"})
    p_queue.enqueue({"id": 7, "requester": "Prof. Smith", "book": "Data Science", "priority": "Faculty"})
    p_queue.enqueue({"id": 8, "requester": "Charlie", "book": "Web Development", "priority": "Student"})
    p_queue.enqueue({"id": 9, "requester": "Prof. Smith", "book": "Machine Learning", "priority": "Faculty"})

    # Display queue
    p_queue.display()

    # Process requests
    print("Processing Requests (Priority Queue) ---")
    while True:
        req = p_queue.dequeue()
        if req:
            print(f"Request processed: {req}")
        else:
            break

```

Output:

```

PS C:\AI Assistant Coding> C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "C:/AI Assistant Coding/bug.py"

=== Priority Queue ===
1. ID: 3, Requester: Charlie (Student), Book: Web Development
2. ID: 5, Requester: Eve (Student), Book: Databases
Queue size: 2

=====
PS C:\AI Assistant Coding>

```

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

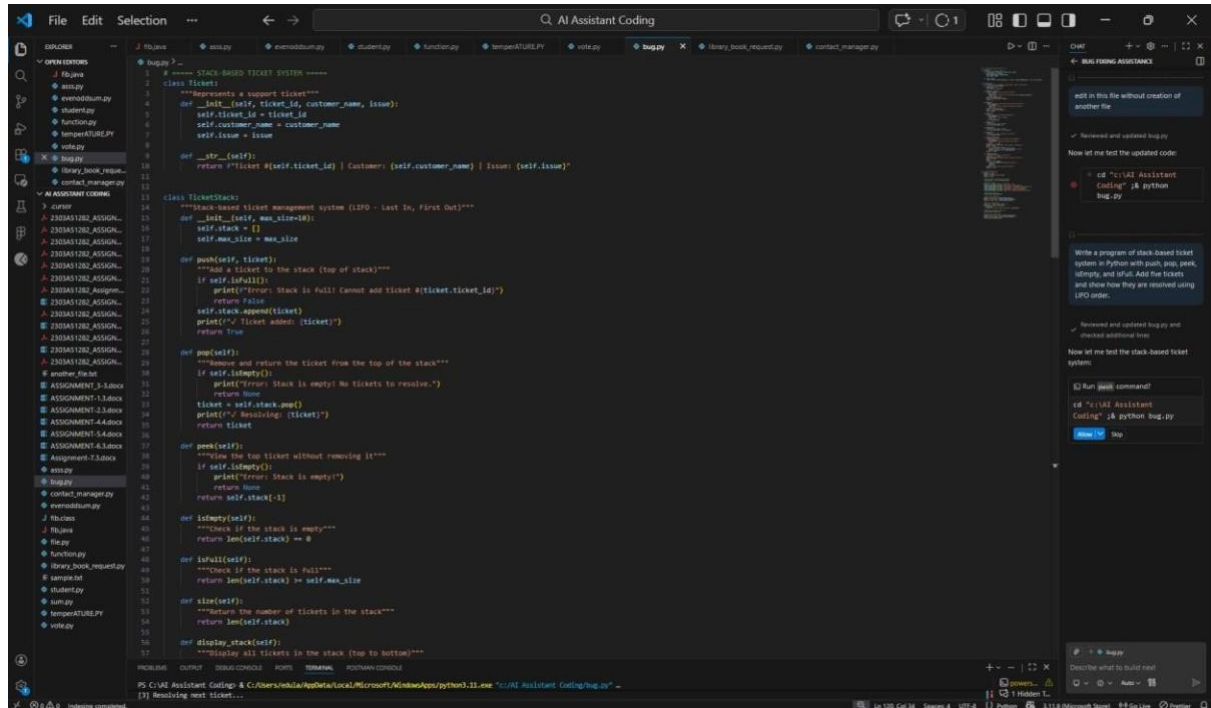
SR University's IT Help Desk receives technical support tickets from students and staff.

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

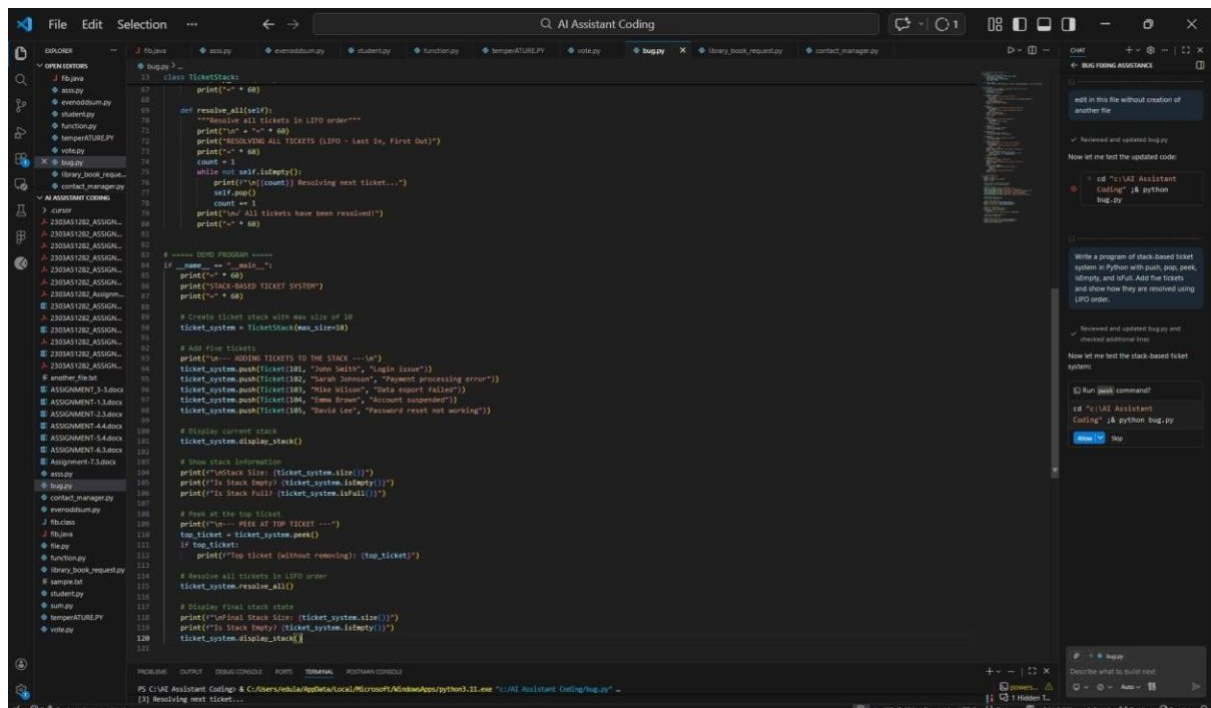
Prompt:

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:

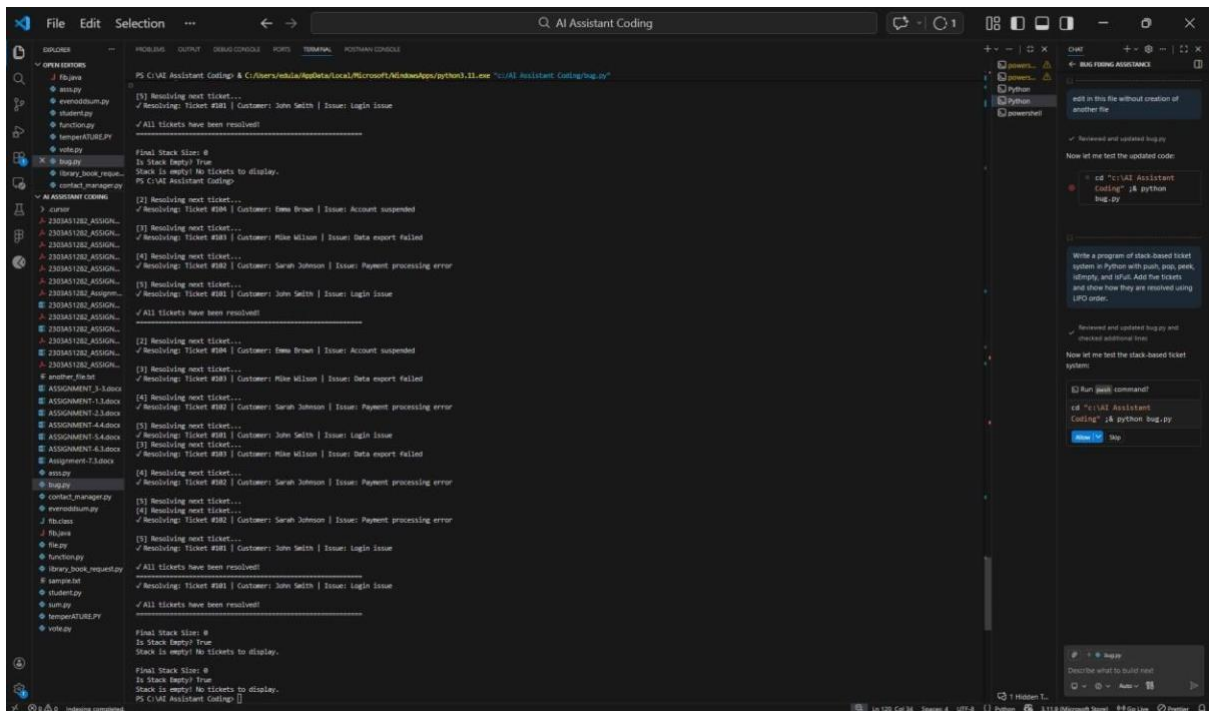


```
1 # ===== STACK-BASED TICKET SYSTEM =====
2 class Ticket:
3     """Represents a support ticket"""
4     def __init__(self, ticket_id, customer_name, issue):
5         self.ticket_id = ticket_id
6         self.customer_name = customer_name
7         self.issue = issue
8
9     def __str__(self):
10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



```
61 def resolve_all(self):
62     """Resolve all tickets in LIFO order"""
63     print("Resolving all tickets in LIFO order")
64     while not self.is_empty():
65         ticket = self.pop()
66         print(f"Resolving next ticket: {ticket}")
67         count += 1
68     print(f"Total tickets resolved: {count}")
69
70 # ===== DEMO PROGRAM =====
71 if __name__ == "__main__":
72     print("===== DEMO =====")
73     # Create ticket stack with max size of 10
74     ticket_system = TicketStack(max_size=10)
75
76     # Add five tickets
77     print("Adding tickets to the stack")
78     ticket_system.push(Ticket(101, "John Doe", "Login issue"))
79     ticket_system.push(Ticket(102, "Sarah Johnson", "Payment processing error"))
80     ticket_system.push(Ticket(103, "Mike Wilson", "Data export failed"))
81     ticket_system.push(Ticket(104, "Anna Brown", "Account suspension"))
82     ticket_system.push(Ticket(105, "David Lee", "Password reset not working"))
83
84     # Display current stack
85     ticket_system.display_stack()
86
87     # Show stack information
88     print(f"Stack Size: {ticket_system.size()}")
89     print(f"Is Stack Empty? {ticket_system.is_empty()}")
90     print(f"Is Stack Full? {ticket_system.is_full()}")
91
92     # Peek at the top ticket
93     print("Peek at top ticket")
94     top_ticket = ticket_system.peek()
95     if top_ticket:
96         print(f"Top ticket (without removing): {top_ticket}")
97
98     # Resolve all tickets in LIFO order
99     ticket_system.resolve_all()
100
101     # Display final stack state
102     print(f"Final Stack Size: {ticket_system.size()}")
103     print(f"Is Stack Empty? {ticket_system.is_empty()}")
104     ticket_system.display_stack()
```

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:

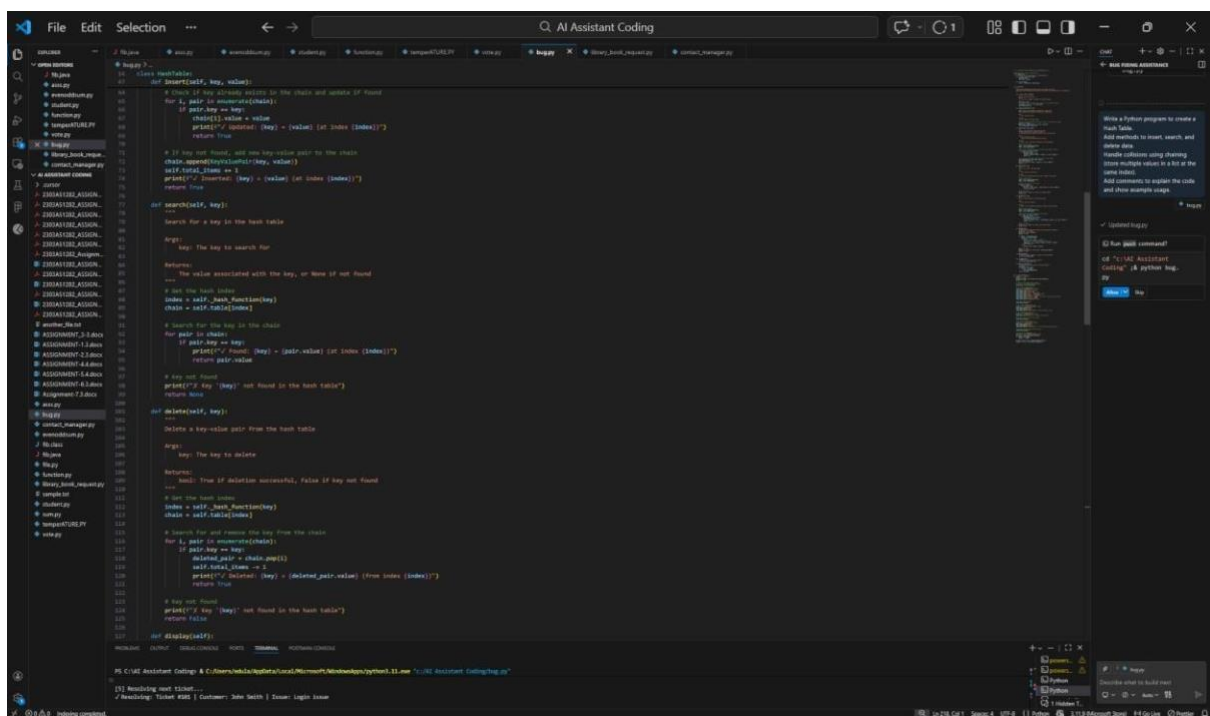
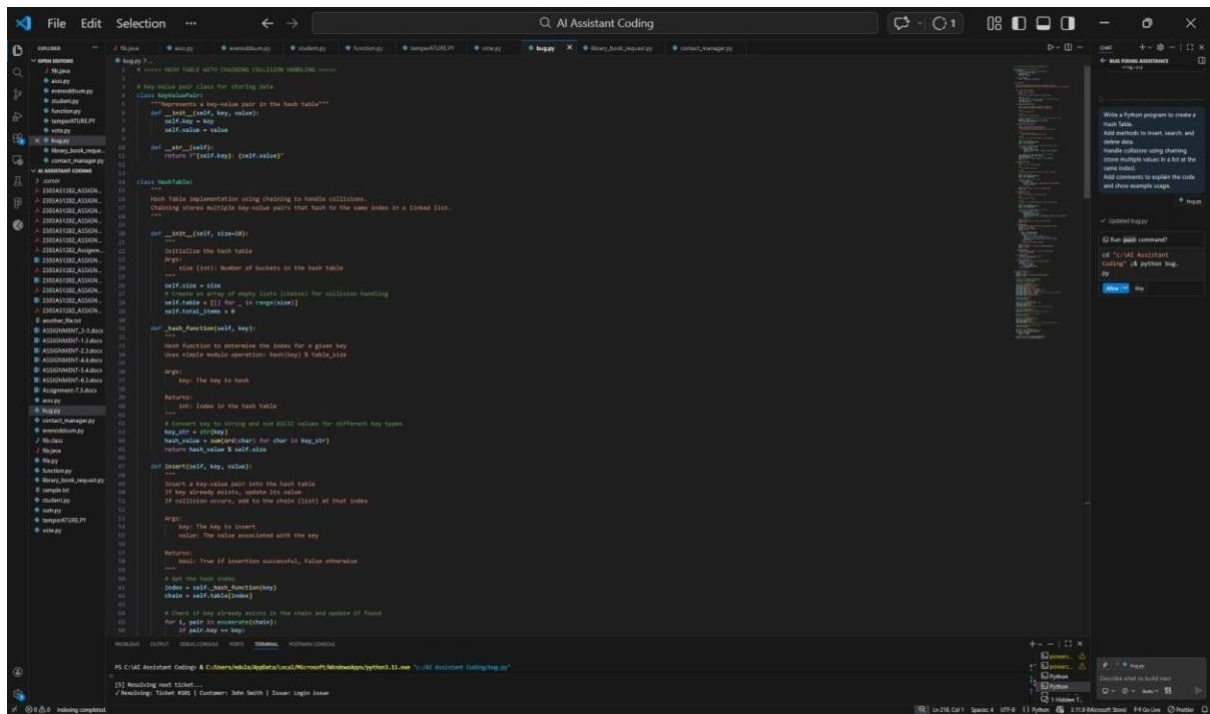
Write a Python program to create a Hash Table.

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:



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:

```
1 # Campus Resource Management System
2 # Features and chosen data structures:
3 # - Attendance: set (O(1) add/check) to track present student IDs
4 # - Event Registration: queue (FIFO queue) for fair registration processing
5 # - Library: dict (hash table) mapping ISBN -> book record for fast lookup
6 # - Bus Schedule: dict of route -> sorted list of departure times (list kept sorted)
7 # - Cafeteria Orders: heap (priority queue) to prioritize faculty over students while preserving arrival order
8
9 Run this file to see a small demo of each feature.
10
11 # ===== Attendance (set) =====
12
13 from collections import deque
14 import heapq
15 import itertools
16 from heapq import heappop
17 from datetime import datetime, timedelta
18
19 class Attendance:
20     """Track attendance using a set for O(1) add/remove/check."""
21     def __init__(self):
22         self.present = set()
23
24     def mark_present(self, student_id):
25         self.present.add(student_id)
26         print(f"Marked present: {student_id}")
27
28     def mark_absent(self, student_id):
29         self.present.discard(student_id)
30         print(f"Marked absent: {student_id}")
31
32     def is_present(self, student_id):
33         return student_id in self.present
34
35     def present_count(self):
36         return len(self.present)
37
38     def list_present(self):
39         return sorted(self.present)
40
41 # ===== Event Registration (FIFO queue) =====
42
43 class EventRegistration:
44     """Register attendees in arrival order using deque."""
45     def __init__(self):
46         self.queue = deque()
47
48     def add_registration(self, attendee_id, name):
49         self.queue.append((attendee_id, name))
50         print(f"Registered: {attendee_id} - {name}")
51
52     def process_registration(self):
53         if not self.queue:
54             print("No registrations to process.")
55             return None
56         attendee = self.queue.popleft()
57         print(f"Processed registration: {attendee[0]} - {attendee[1]}")
58         return attendee
59
60     def pending_count(self):
61         return len(self.queue)
62
63 # ===== Library Book Request (dict) =====
64
65 class LibraryBookRequest:
66     """Manage book requests using a dict for ISBN mapping."""
67     def __init__(self):
68         self.requests = {}
69
70     def add_request(self, isbn, title, author):
71         self.requests[isbn] = (title, author)
72         print(f"Added request for ISBN {isbn}: {title} by {author}")
73
74     def get_request(self, isbn):
75         return self.requests.get(isbn)
76
77 # ===== Bus Schedule (dict of sorted lists) =====
78
79 class BusSchedule:
80     """Manage bus schedules using a dict of sorted lists for routes and times."""
81     def __init__(self):
82         self.routes = {}
83
84     def add_route(self, route_id, times):
85         self.routes[route_id] = sorted(times)
86         print(f"Added route {route_id} with times {times}")
87
88     def get_route(self, route_id):
89         return self.routes.get(route_id)
90
91 # ===== Cafeteria Order Queue (heap) =====
92
93 class CafeteriaOrderQueue:
94     """Manage cafeteria orders using a heap for prioritization (faculty over students)."""
95     def __init__(self):
96         self.orders = []
97
98     def add_order(self, priority, item):
99         heapq.heappush(self.orders, (priority, item))
100         print(f"Added order with priority {priority}: {item}")
101
102     def get_order(self):
103         if not self.orders:
104             print("No orders in queue.")
105             return None
106         priority, item = heapq.heappop(self.orders)
107         print(f"Processed order with priority {priority}: {item}")
108         return item
109
110 # ===== Main Demo =====
111
112 def main():
113     # Attendance Demo
114     attendance = Attendance()
115     attendance.mark_present(12345)
116     attendance.mark_absent(67890)
117     print(f"Present count: {attendance.present_count()}")
118     print(f"List of present students: {attendance.list_present()}")
119
120     # Event Registration Demo
121     reg = EventRegistration()
122     reg.add_registration(1, "Alice")
123     reg.add_registration(2, "Bob")
124     reg.add_registration(3, "Charlie")
125     reg.process_registration()
126     reg.process_registration()
127     reg.process_registration()
128     print(f"Pending count: {reg.pending_count()}")
129
130     # Library Book Request Demo
131     lib = LibraryBookRequest()
132     lib.add_request("ISBN123", "The Great Gatsby", "F. Scott Fitzgerald")
133     lib.add_request("ISBN456", "1984", "George Orwell")
134     lib.get_request("ISBN123")
135
136     # Bus Schedule Demo
137     bus = BusSchedule()
138     bus.add_route("Route A", [10:00, 11:00, 12:00, 13:00, 14:00, 15:00])
139     bus.add_route("Route B", [9:00, 10:00, 11:00, 12:00, 13:00, 14:00])
140     bus.get_route("Route A")
141     bus.get_route("Route B")
142
143     # Cafeteria Order Queue Demo
144     cafeteria = CafeteriaOrderQueue()
145     cafeteria.add_order(1, "Faculty Sandwich")
146     cafeteria.add_order(2, "Student Smoothie")
147     cafeteria.add_order(3, "Faculty Salad")
148     cafeteria.get_order()
149     cafeteria.get_order()
150     cafeteria.get_order()
151
152 if __name__ == "__main__":
153     main()
```



```
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: S001
Marked present: S002
Marked present: S010
Present list: ['S001', 'S002', 'S010']
Is S002 present? True
Marked absent: S002
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).
S001 borrowed Clean Code
S003 borrowed Clean Code
No copies available.
Available books: [('978-0135166307', 'Clean Code', 0), ('978-1491958296', 'Fluent Python', 1)]
S001 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 18:42:24.367227
Added bus time for Route A: 2026-02-18 18:57:24.367227
Added bus time for Route B: 2026-02-18 18:39:24.367227
Next Route A bus: 2026-02-18 18:42:24.367227
Route A schedule: [datetime.datetime(2026, 2, 18, 18, 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.