# AI ASSISTANT CODING
## ASSIGNMENT - 11.3

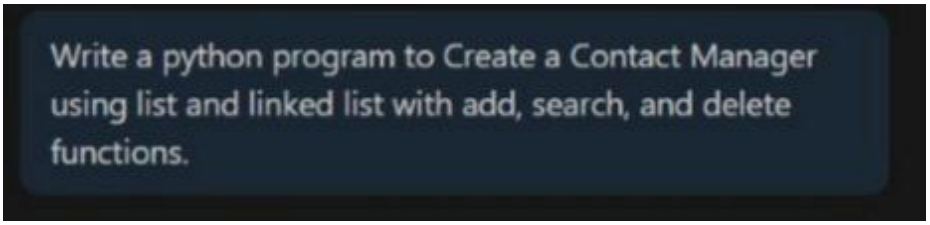
**Name:** Aman Sarkar
**Ht.no:** 2303A51273
**Batch:** 05

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

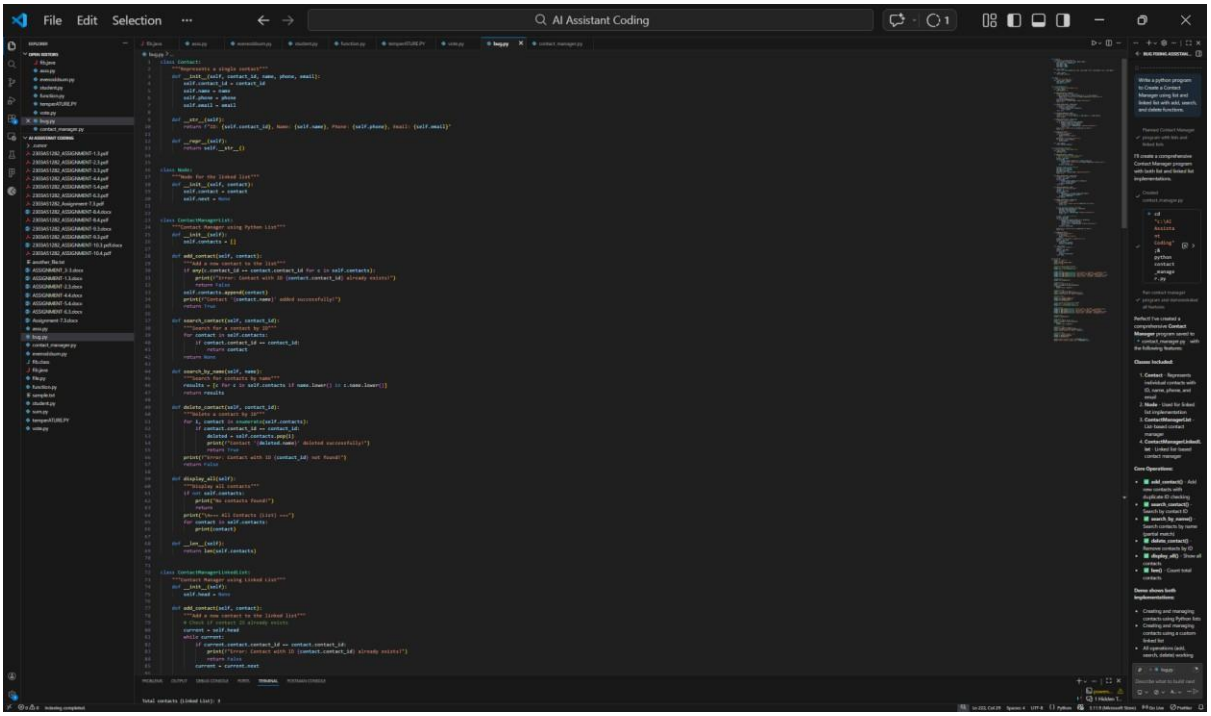
Scenario
SR University's student club requires a simple Contact Manager Application to store members' names and phone numbers. The system should support efficient addition, searching, and deletion of contacts.

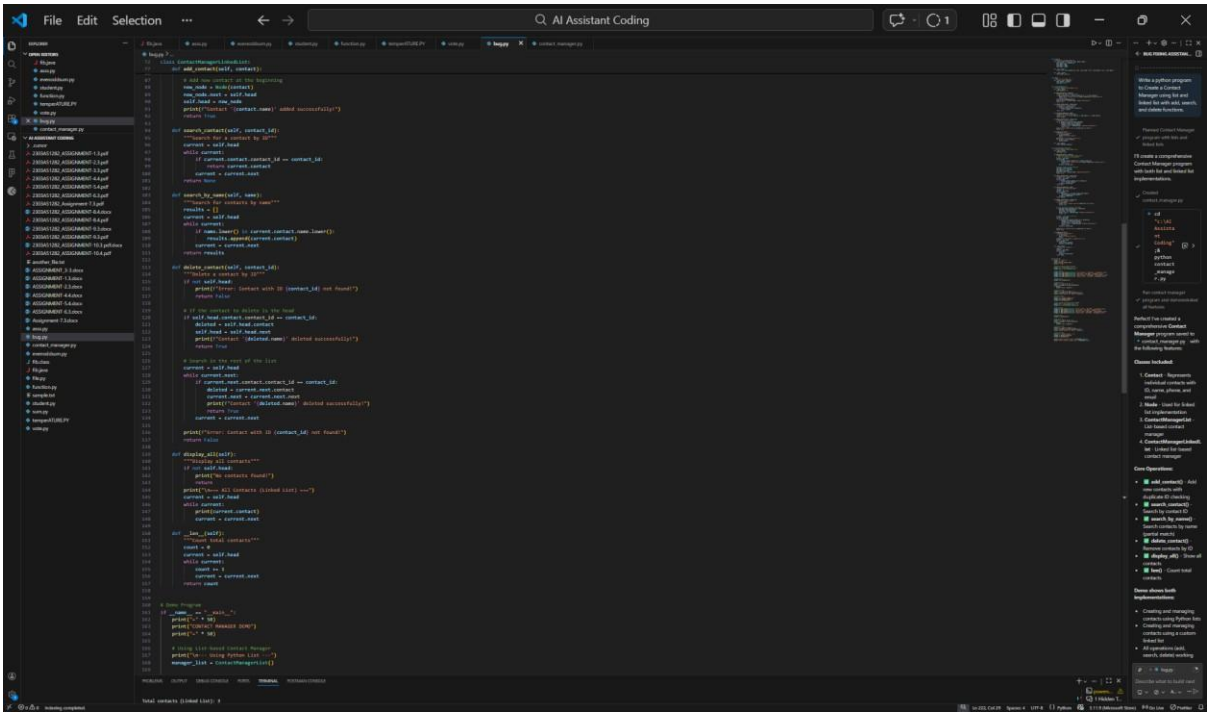
## Prompt:

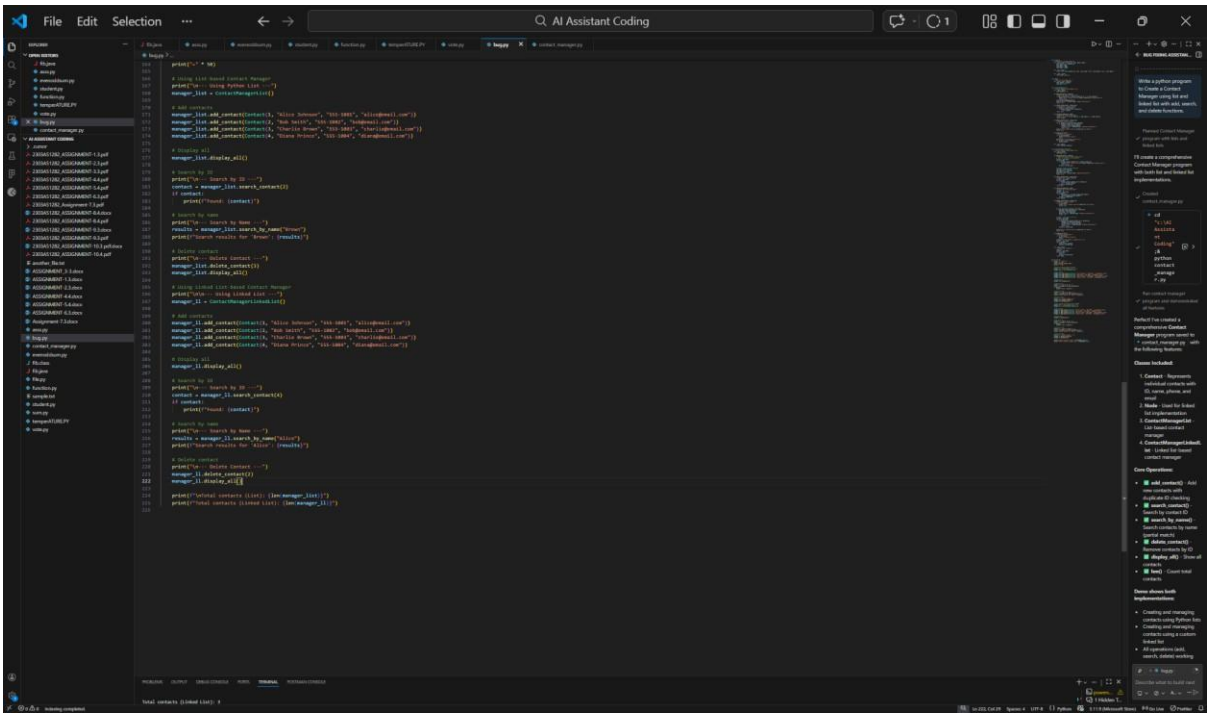
Write a python program to Create a Contact Manager using list and linked list with add, search, and delete functions.



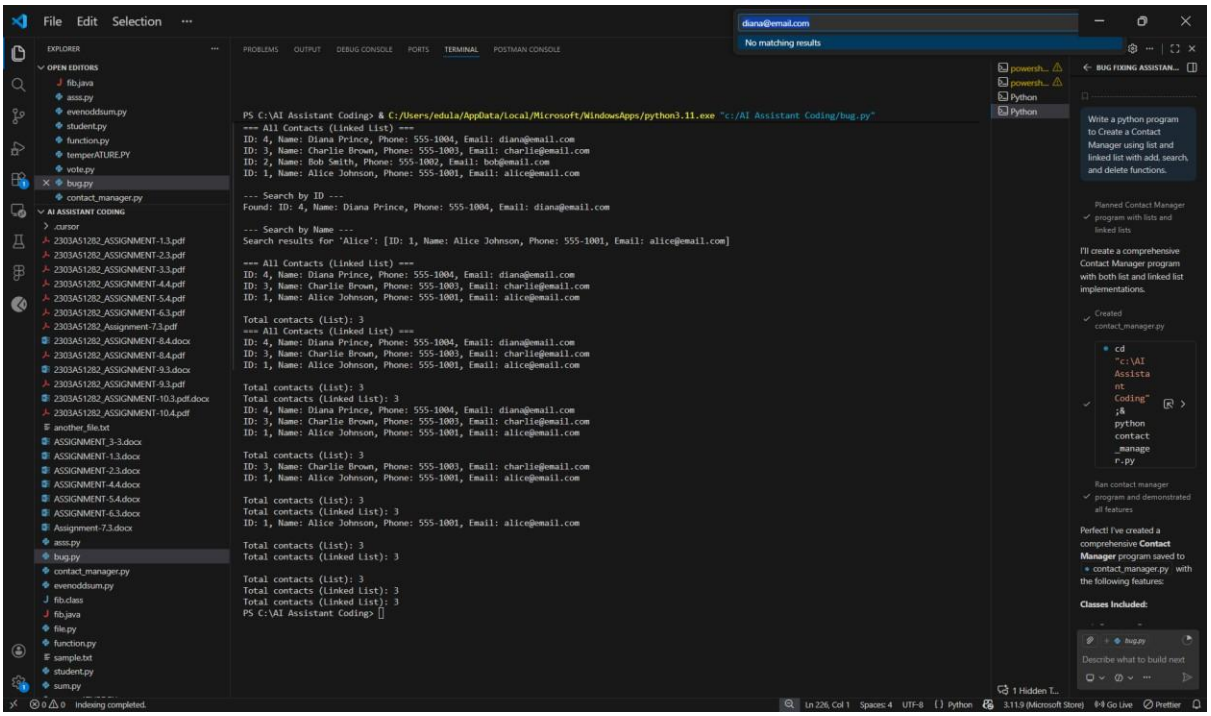
# Code:


File Edit Selection
AI Assistant Coding


File Edit Selection
AI Assistant Coding



# Output:




# 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.

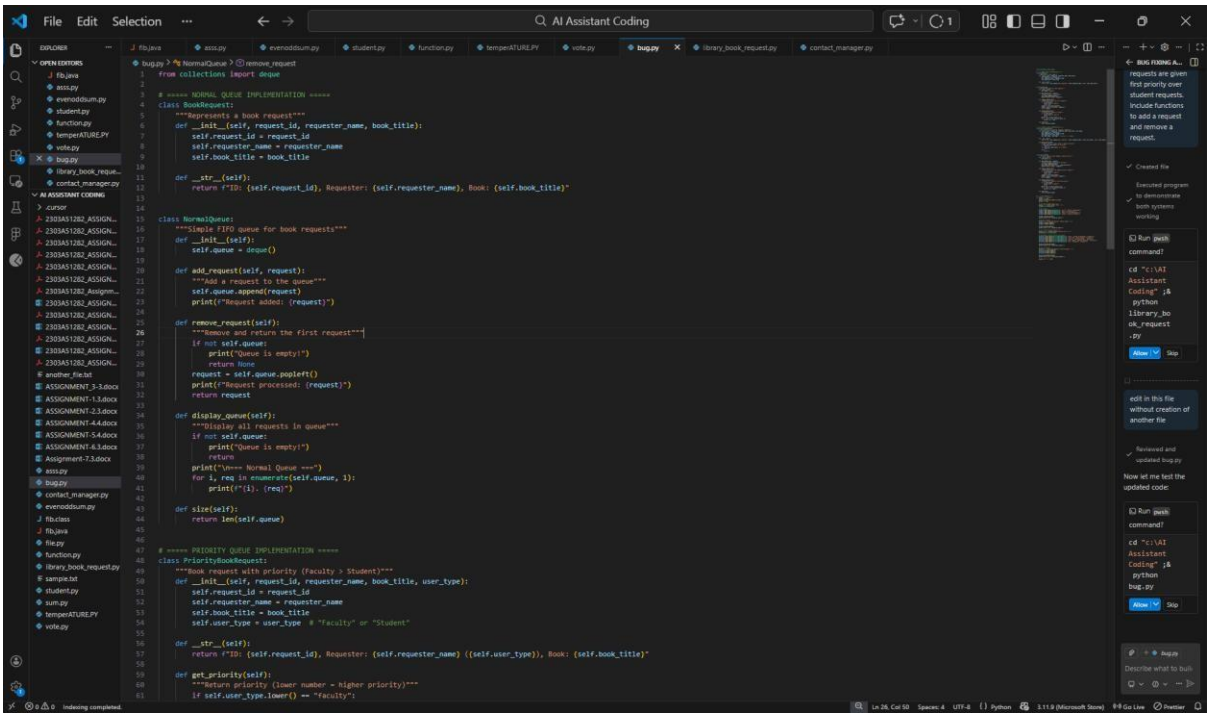
**Task 2:** Library Book Search System (Queues & Priority Queues)
Scenario
The SRU Library manages book borrow requests. Students and faculty submit requests, but faculty requests must be prioritized over student requests.

**Prompt:**

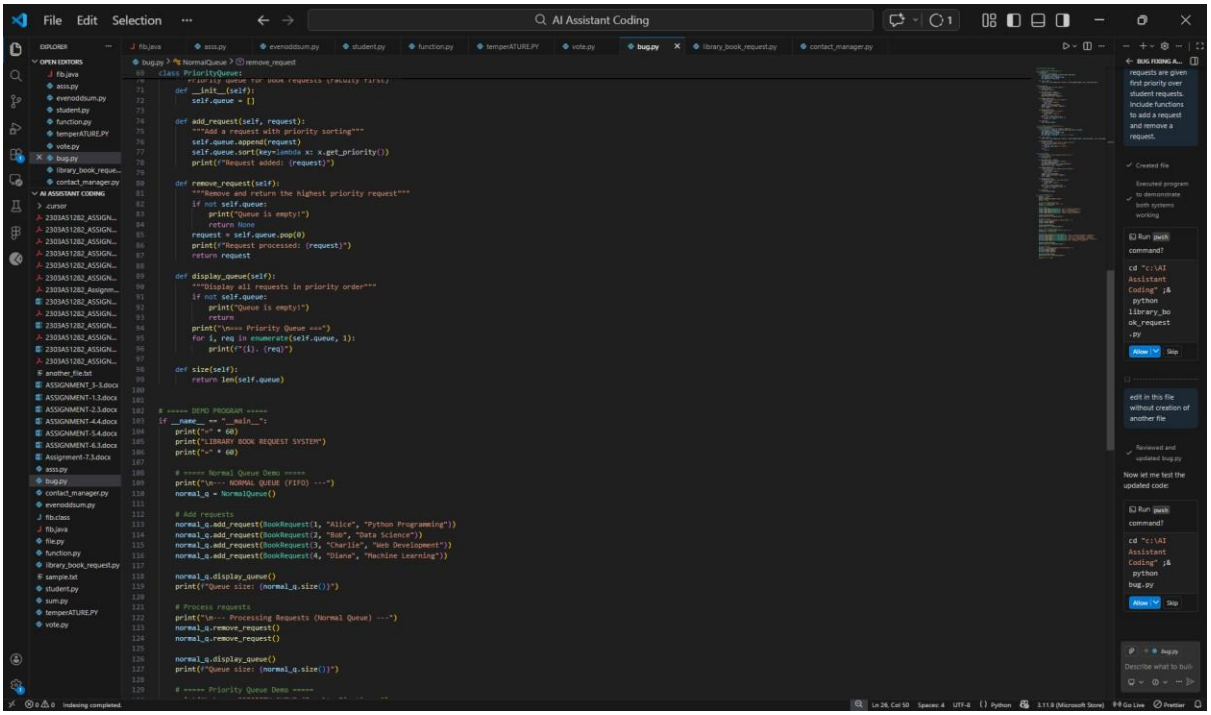
Write a Python program for a library book request system.First, make a normal queue where requests are handled in the order they come.Then, make another version where faculty requests are given first priority over student requests.Include functions to add a request and remove a request.

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.

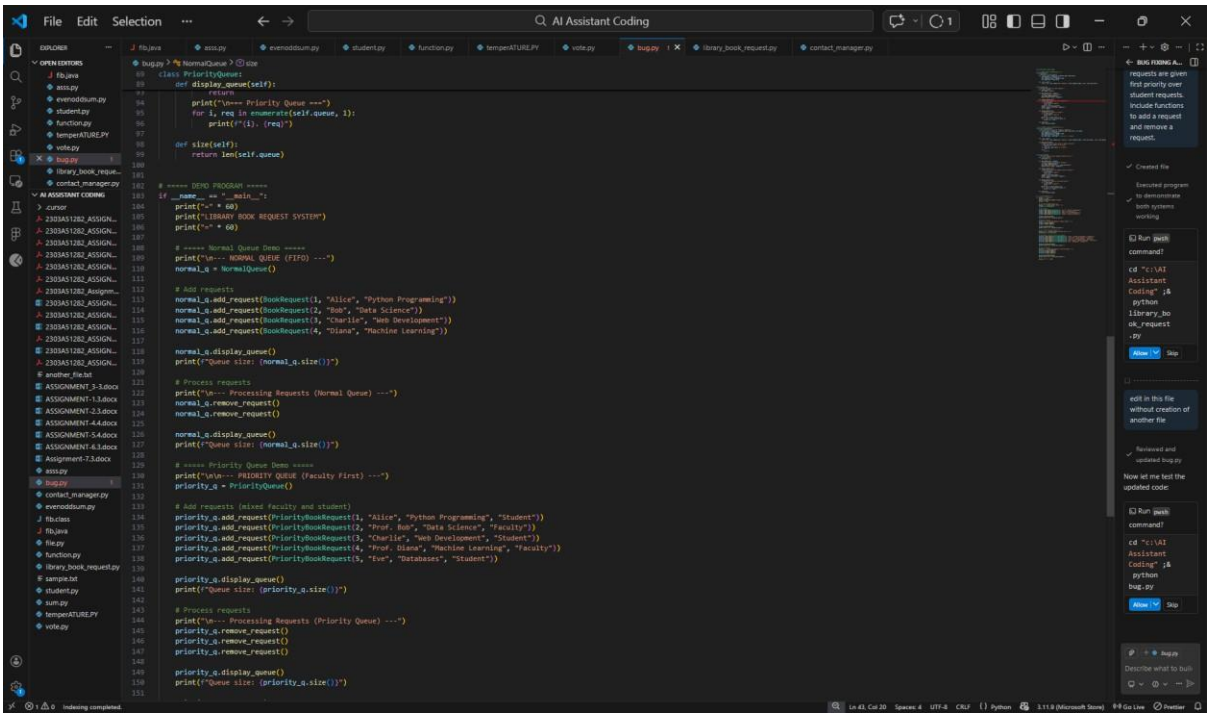
# Code:



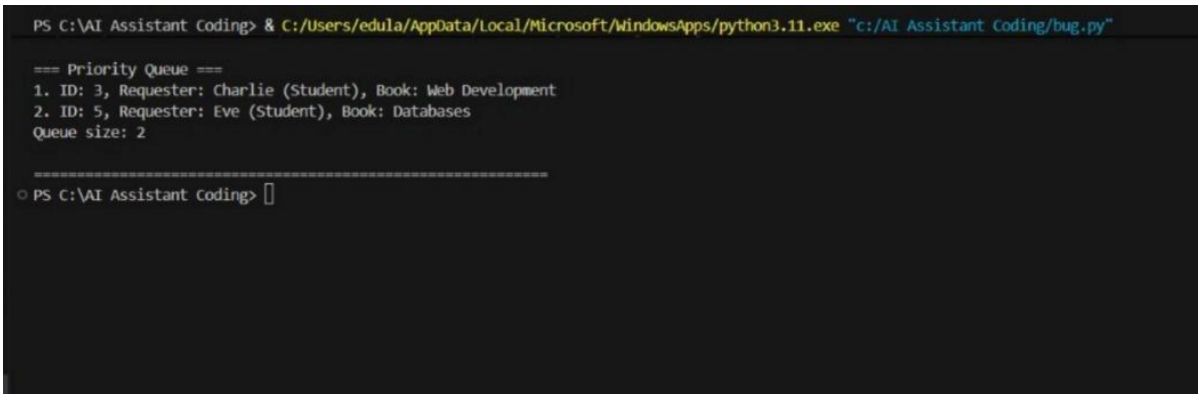
```python
from collections import deque

# ===== NORMAL QUEUE IMPLEMENTATION =====
class BookRequest:
    """Represents a book request"""
    def __init__(self, request_id, requester_name, book_title):
        self.request_id = request_id
        self.requester_name = requester_name
        self.book_title = book_title

    def __str__(self):
        return f"ID: {self.request_id}, Requester: {self.requester_name}, Book: {self.book_title}"


class NormalQueue:
    """Simple FIFO queue for book requests"""
    def __init__(self):
        self.queue = deque()

    def add_request(self, request):
        """Add a request to the queue"""
        self.queue.append(request)
        print(f"Request added: {request}")

    def remove_request(self):
        """Remove and return the first request"""
        if not self.queue:
            print("Queue is empty!")
            return None
        request = self.queue.popleft()
        print(f"Request processed: {request}")
        return request

    def display_queue(self):
        """Display all requests in queue"""
        if not self.queue:
            print("Queue is empty!")
            return
        print("\n=== Normal Queue ===")
        for i, req in enumerate(self.queue, 1):
            print(f"{i}. {req}")

    def size(self):
        return len(self.queue)


# ===== PRIORITY QUEUE IMPLEMENTATION =====
class PriorityBookRequest:
    """Book request with priority (Faculty > Student)"""
    def __init__(self, request_id, requester_name, book_title, user_type):
        self.request_id = request_id
        self.requester_name = requester_name
        self.book_title = book_title
        self.user_type = user_type  # "Faculty" or "Student"

    def __str__(self):
        return f"ID: {self.request_id}, Requester: {self.requester_name} ({self.user_type}), Book: {self.book_title}"

    def get_priority(self):
        """Return priority (lower number = higher priority)"""
        if self.user_type.lower() == "faculty":
```



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

    def add_request(self, request):
        """Add a request with priority sorting"""
        self.queue.append(request)
        self.queue.sort(key=lambda x: x.get_priority())
        print(f"Request added: {request}")

    def remove_request(self):
        """Remove and return the highest priority request"""
        if not self.queue:
            print("Queue is empty!")
            return None
        request = self.queue.pop(0)
        print(f"Request processed: {request}")
        return request

    def display_queue(self):
        """Display all requests in priority order"""
        if not self.queue:
            print("Queue is empty!")
            return
        print("\n=== Priority Queue ===")
        for i, req in enumerate(self.queue, 1):
            print(f"{i}. {req}")

    def size(self):
        return len(self.queue)


# ===== DEMO PROGRAM =====
if __name__ == "__main__":
    print("=" * 60)
    print("LIBRARY BOOK REQUEST SYSTEM")
    print("=" * 60)

    # ===== Normal Queue Demo =====
    print("\n--- NORMAL QUEUE (FIFO) ---")
    normal_q = NormalQueue()

    # Add requests
    normal_q.add_request(BookRequest(1, "Alice", "Python Programming"))
    normal_q.add_request(BookRequest(2, "Bob", "Data Science"))
    normal_q.add_request(BookRequest(3, "Charlie", "Web Development"))
    normal_q.add_request(BookRequest(4, "Diana", "Machine Learning"))

    normal_q.display_queue()
    print(f"Queue size: {normal_q.size()}")

    # Process requests
    print("\n--- Processing Requests (Normal Queue) ---")
    normal_q.remove_request()
    normal_q.remove_request()

    normal_q.display_queue()
    print(f"Queue size: {normal_q.size()}")

    # ===== Priority Queue Demo =====
```




## Output:




## Explanation:

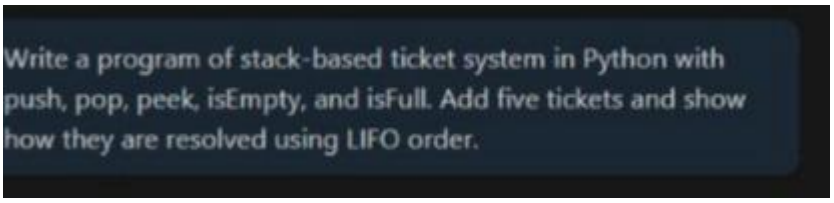
- Queue (FIFO) → First request comes, first served.(If a student requests first, they get the book first.)
- Priority Queue → Faculty requests are served before students, even if they come later.
- enqueue() → Adds a request to the system.

- dequeue() → Removes and processes the next request.

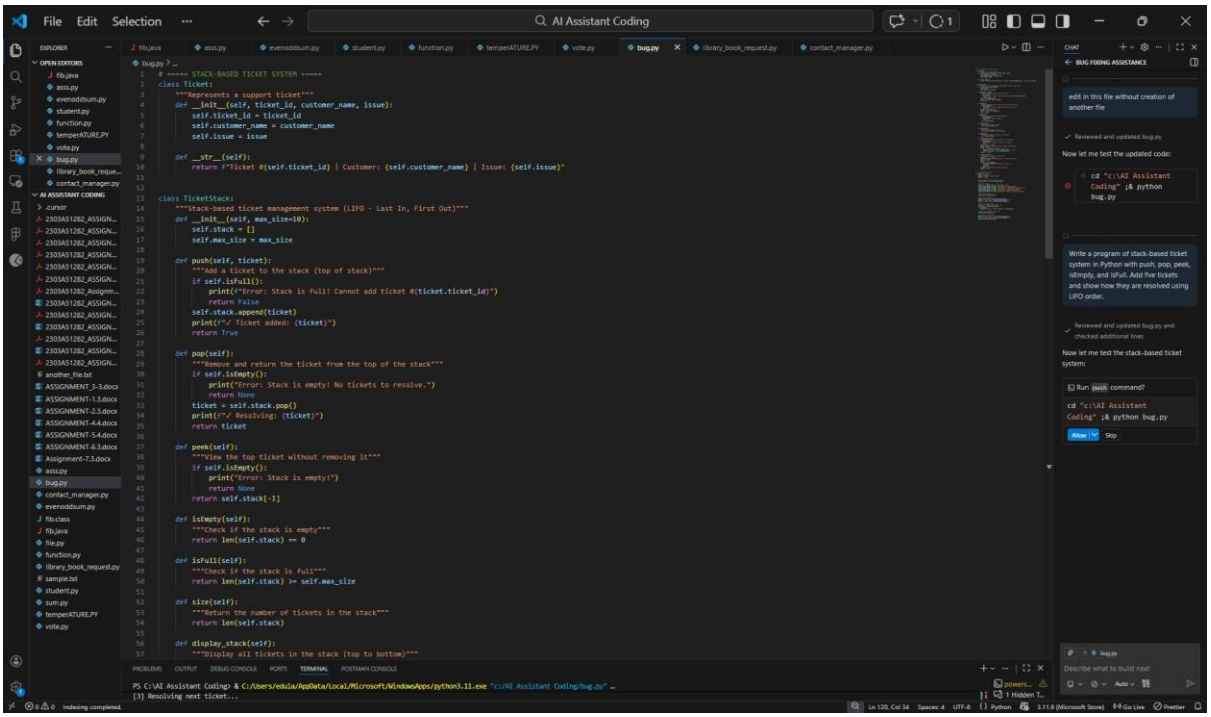
**Task 3:** Emergency Help Desk (Stack Implementation)
Scenario
SR University's IT Help Desk receives technical support tickets from students and staff. While tickets are received sequentially, issue escalation follows a Last-In, First-Out (LIFO) approach.

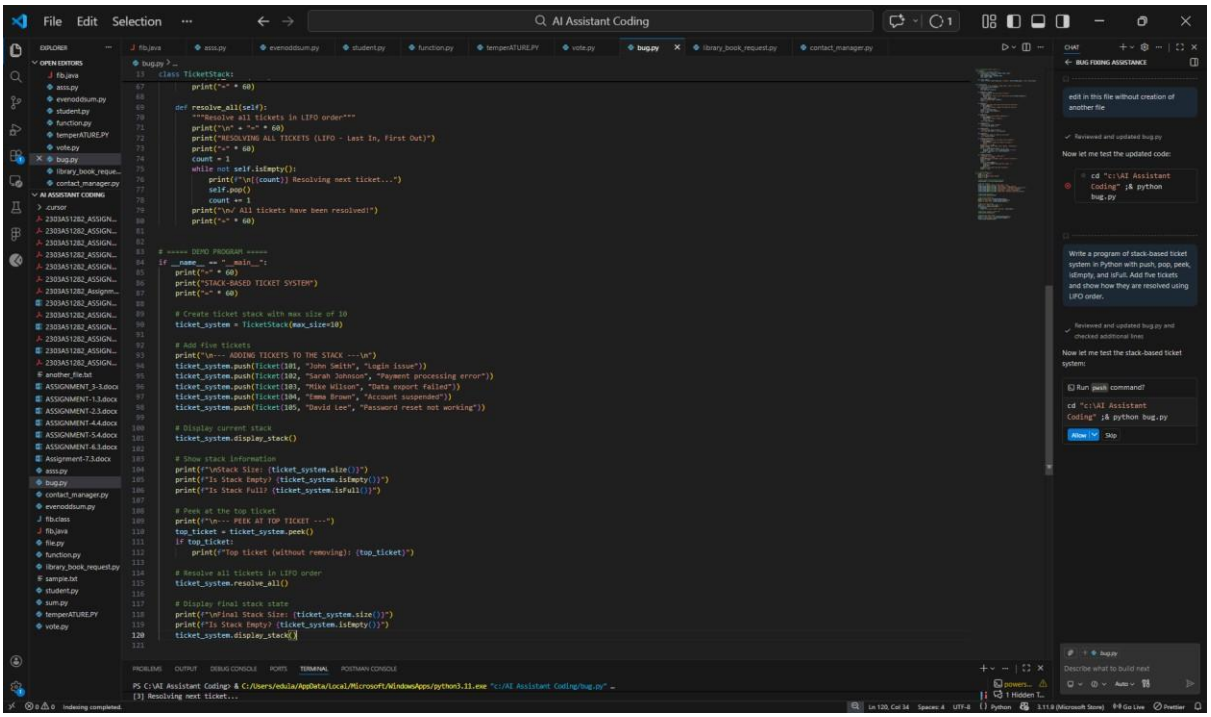
**Prompt:**

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.

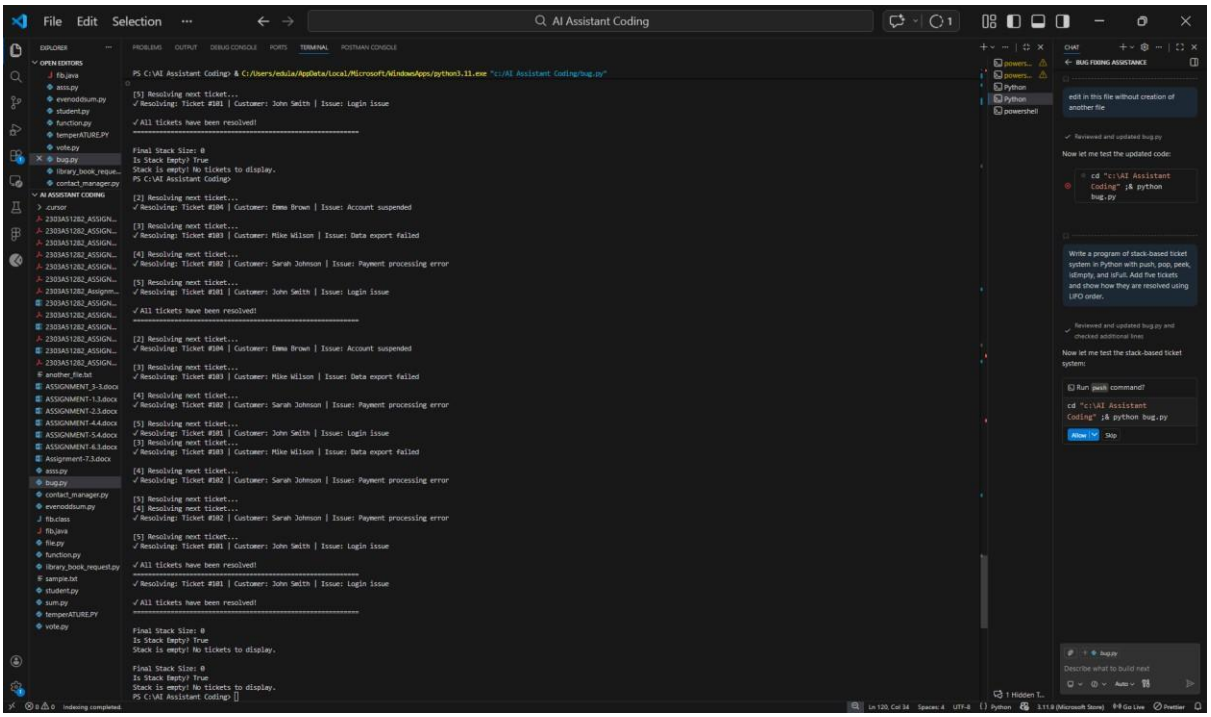



**Code:**


File Edit Selection
AI Assistant Coding
bug.py
# ===== STACK-BASED TICKET SYSTEM =====
class Ticket:
"""Represents a support ticket"""
def __init__(self, ticket_id, customer_name, issue):
self.ticket_id = ticket_id
self.customer_name = customer_name
self.issue = issue
def __str__(self):
return f"Ticket #{self.ticket_id} | Customer: {self.customer_name} | Issue: {self.issue}"
class TicketStack:
"""Stack-based ticket management system (LIFO - Last In, First Out)"""
def __init__(self, max_size=10):
self.stack = []
self.max_size = max_size
def push(self, ticket):
"""Add a ticket to the stack (top of stack)"""
if self.isFull():
print(f"Error: Stack is Full! Cannot add ticket #{ticket.ticket_id}")
return False
self.stack.append(ticket)
print(f"✓ Ticket added: {ticket}")
return True
def pop(self):
"""Remove and return the ticket from the top of the stack"""
if self.isEmpty():
print("Error: Stack is empty! No tickets to resolve.")
return None
ticket = self.stack.pop()
print(f"✓ Resolving: {ticket}")
return ticket
def peek(self):
"""View the top ticket without removing it"""
if self.isEmpty():
print("Error: Stack is empty!")
return None
return self.stack[-1]
def isEmpty(self):
"""Check if the stack is empty"""
return len(self.stack) == 0
def isFull(self):
"""Check if the stack is full"""
return len(self.stack) >= self.max_size
def size(self):
"""Return the number of tickets in the stack"""
return len(self.stack)
def display_stack(self):
"""Display all tickets in the stack (top to bottom)"""
PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bug.py" …
[3] Resolving next ticket...


File Edit Selection
AI Assistant Coding
bug.py
def display_stack(self):
"""Display all tickets in the stack (top to bottom)"""
print("=" * 60)
def resolve_all(self):
"""Resolve all tickets in LIFO order"""
print("\n" + "=" * 60)
print("RESOLVING ALL TICKETS (LIFO - Last In, First Out)")
print("=" * 60)
count = 1
while not self.isEmpty():
print(f"\n[{count}] Resolving next ticket...")
self.pop()
count += 1
print("\n✓ All tickets have been resolved!")
print("=" * 60)
# ===== DEMO PROGRAM =====
if __name__ == "__main__":
print("=" * 60)
print("STACK-BASED TICKET SYSTEM")
print("=" * 60)
# Create ticket stack with max size of 10
ticket_system = TicketStack(max_size=10)
# Add five tickets
print("\n--- ADDING TICKETS TO THE STACK ---\n")
ticket_system.push(Ticket(101, "John Smith", "Login issue"))
ticket_system.push(Ticket(102, "Sarah Johnson", "Payment processing error"))
ticket_system.push(Ticket(103, "Mike Wilson", "Data export failed"))
ticket_system.push(Ticket(104, "Emma Brown", "Account suspended"))
ticket_system.push(Ticket(105, "David Lee", "Password reset not working"))
# Display current stack
ticket_system.display_stack()
# Show stack information
print(f"\nStack Size: {ticket_system.size()}")
print(f"Is Stack Empty? {ticket_system.isEmpty()}")
print(f"Is Stack Full? {ticket_system.isFull()}")
# Peek at the top ticket
print("\n--- PEEK AT TOP TICKET ---")
top_ticket = ticket_system.peek()
if top_ticket:
print(f"Top ticket (without removing): {top_ticket}")
# Resolve all tickets in LIFO order
ticket_system.resolve_all()
# Display final stack state
print(f"\nFinal Stack Size: {ticket_system.size()}")
print(f"Is Stack Empty? {ticket_system.isEmpty()}")
ticket_system.display_stack()
PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bug.py" …
[3] Resolving next ticket...

# 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.

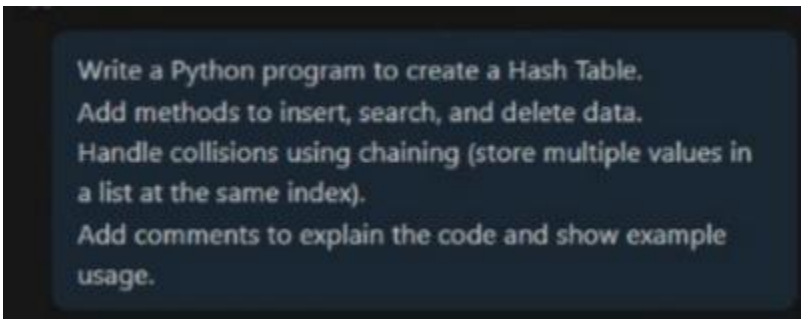
**Task 4:** Hash Table

**Objective**

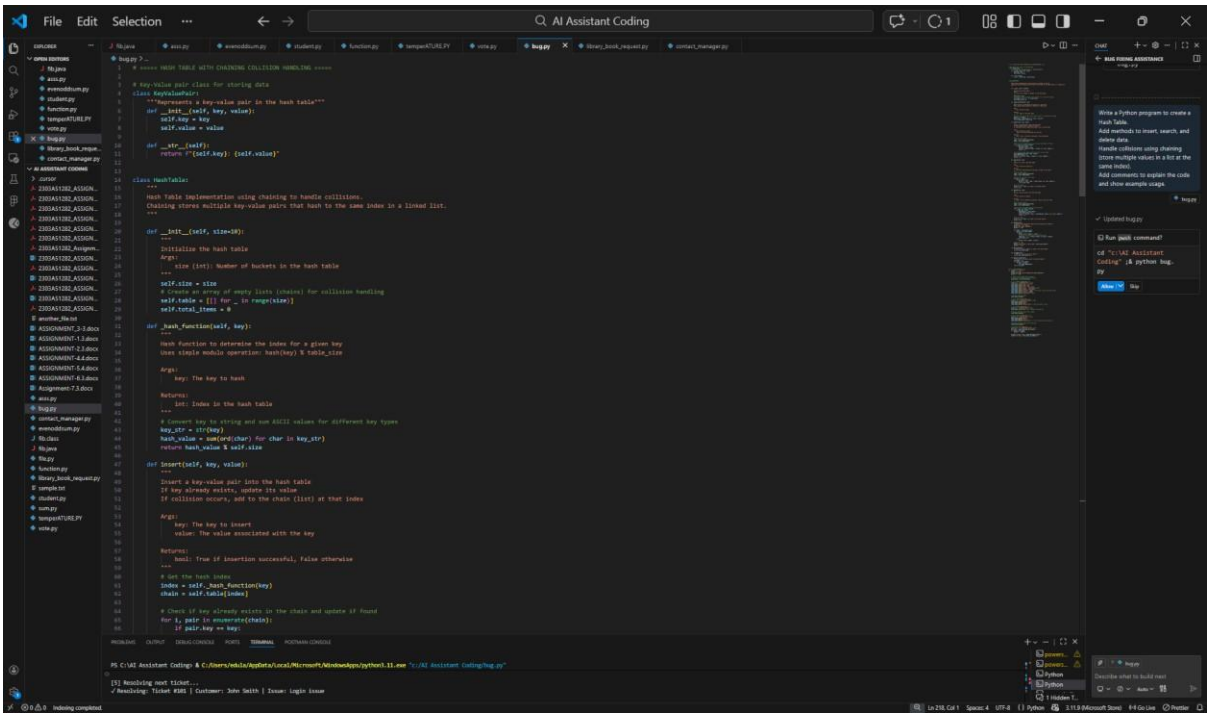
To implement a Hash Table and understand collision handling.

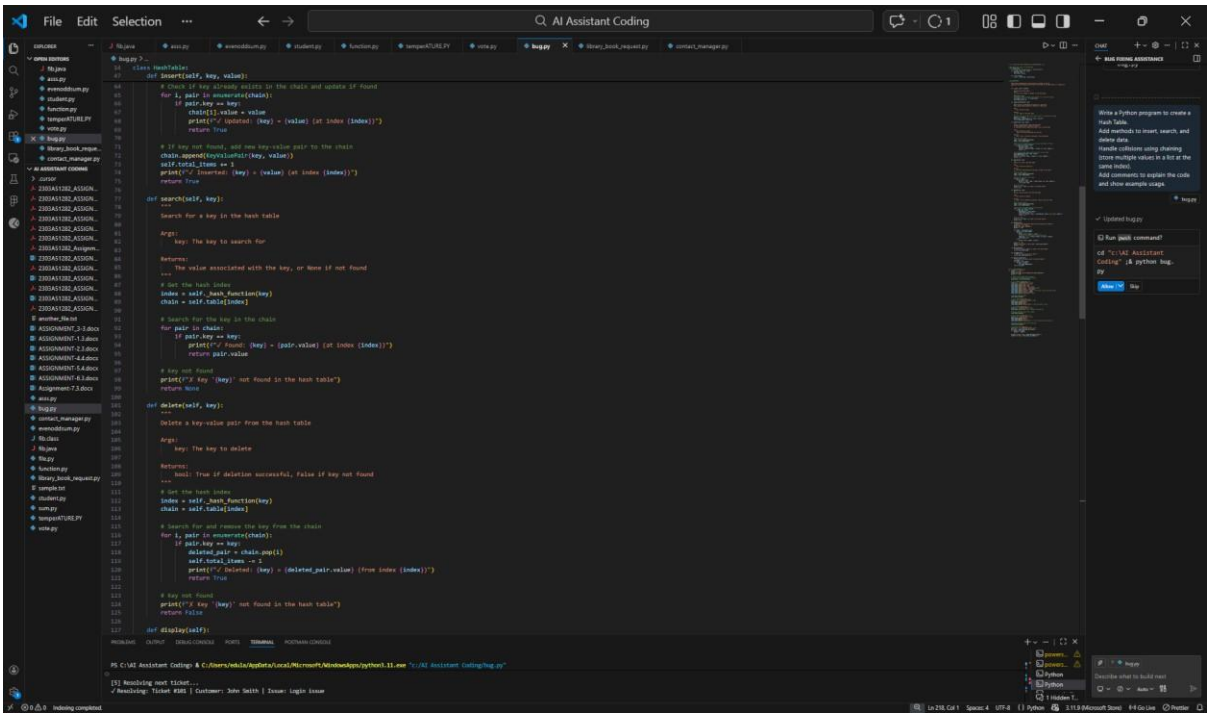
**Prompt:**

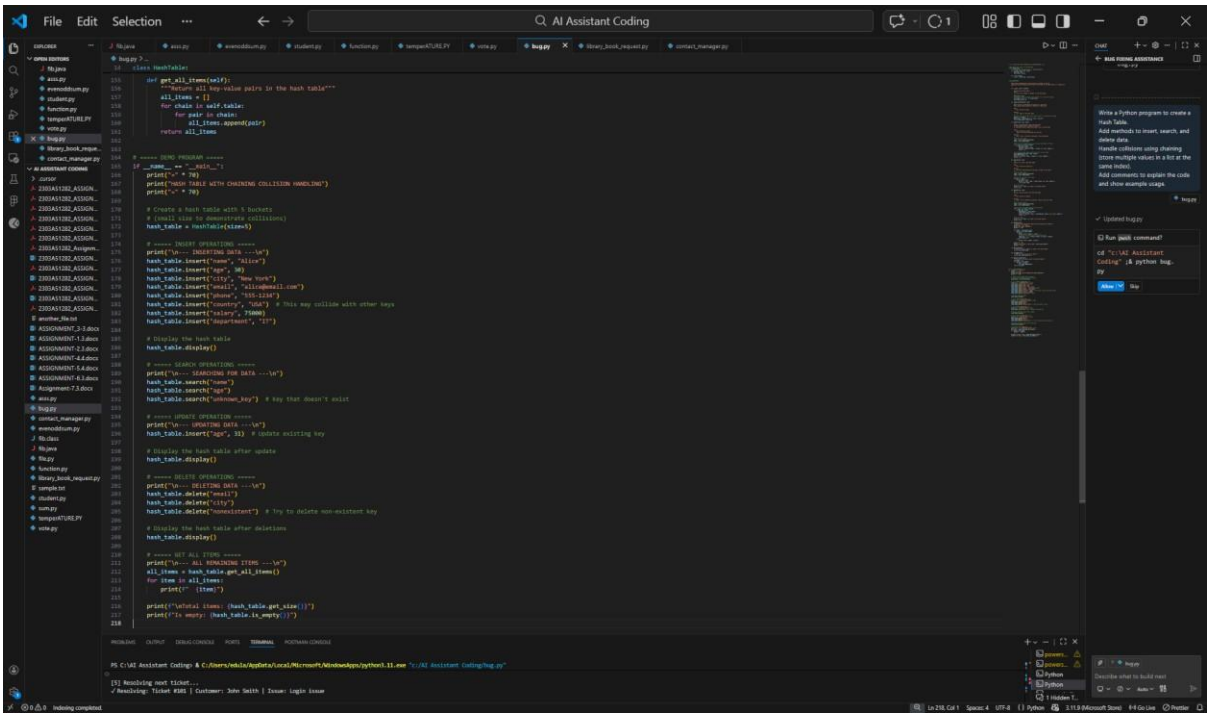
Write a Python program to create a Hash Table.
Add methods to insert, search, and delete data.
Handle collisions using chaining (store multiple values in a list at the same index).
Add comments to explain the code and show example usage.



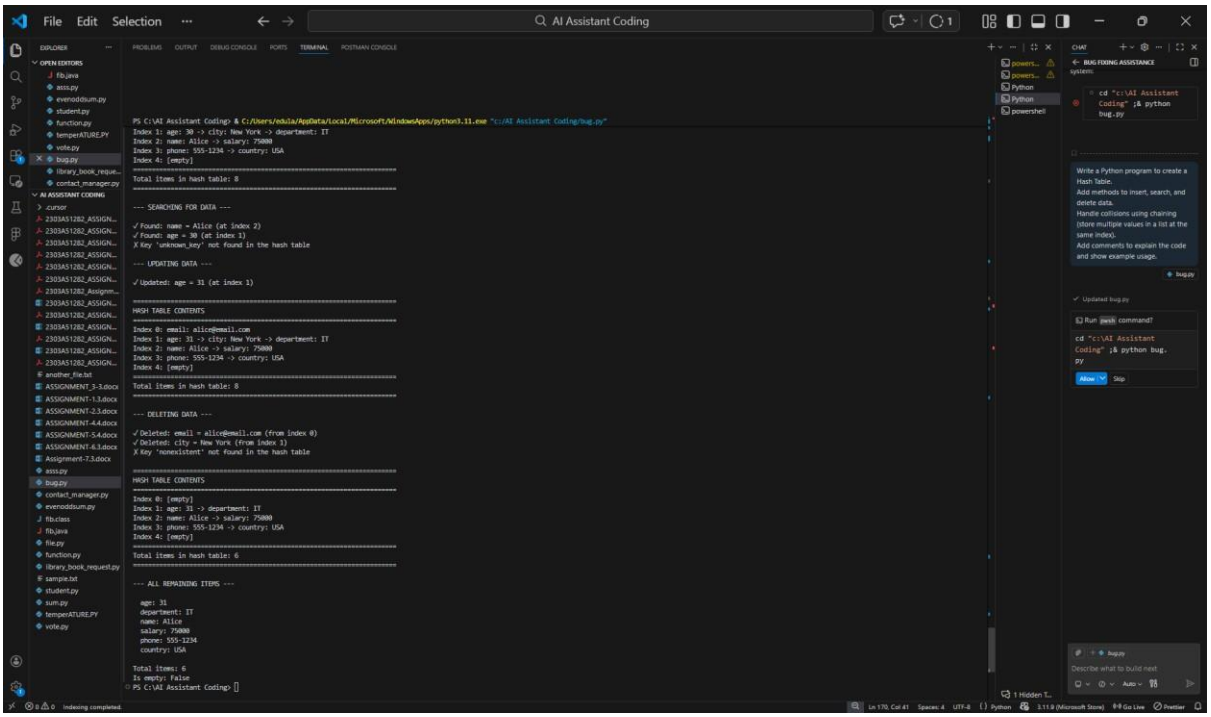

## Code:




File Edit Selection
AI Assistant Coding


File Edit Selection
AI Assistant Coding

# Output:




## Explanation:

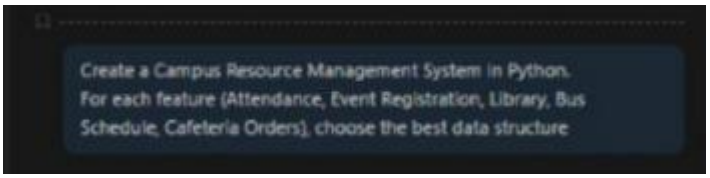
- A Hash Table stores data using a key and value.
- A hash function decides where to store the data.
- Sometimes two keys go to the same place. This is called a collision.
- To solve collisions, we use chaining, meaning we store multiple items in a list at the same index.
- The program should allow adding,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

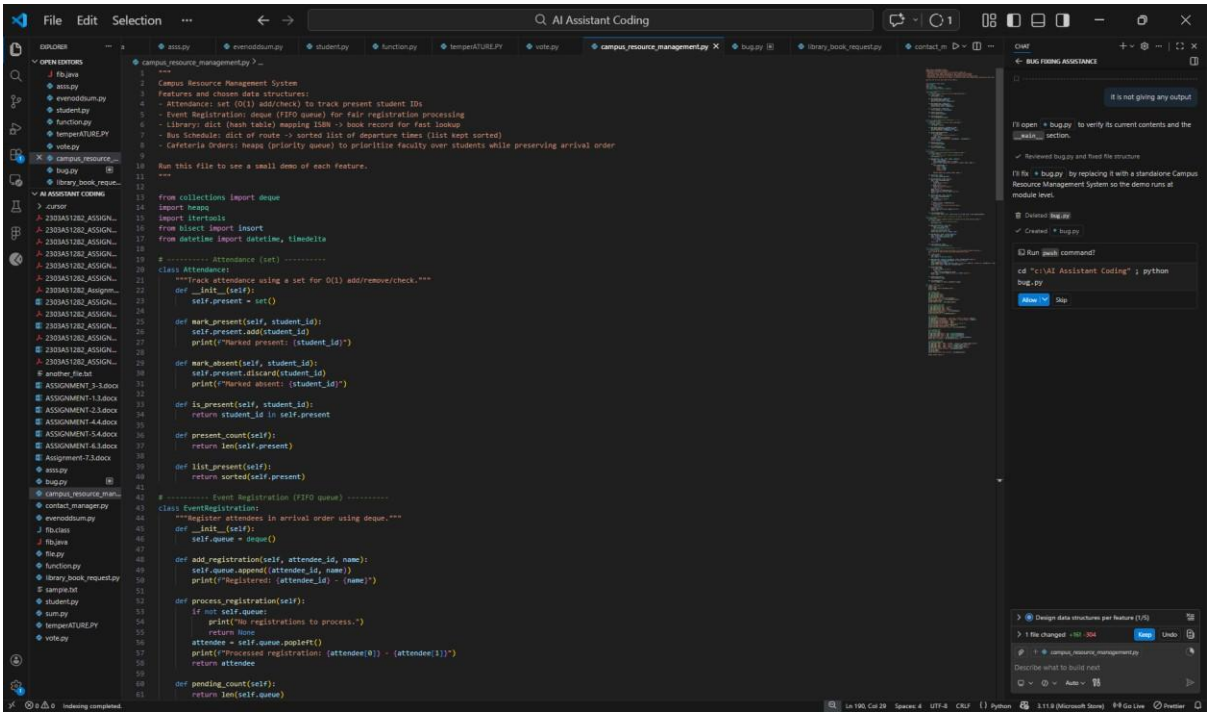
- Event Registration System
- Library Book Borrowing
- Bus Scheduling System
- Cafeteria Order Queue

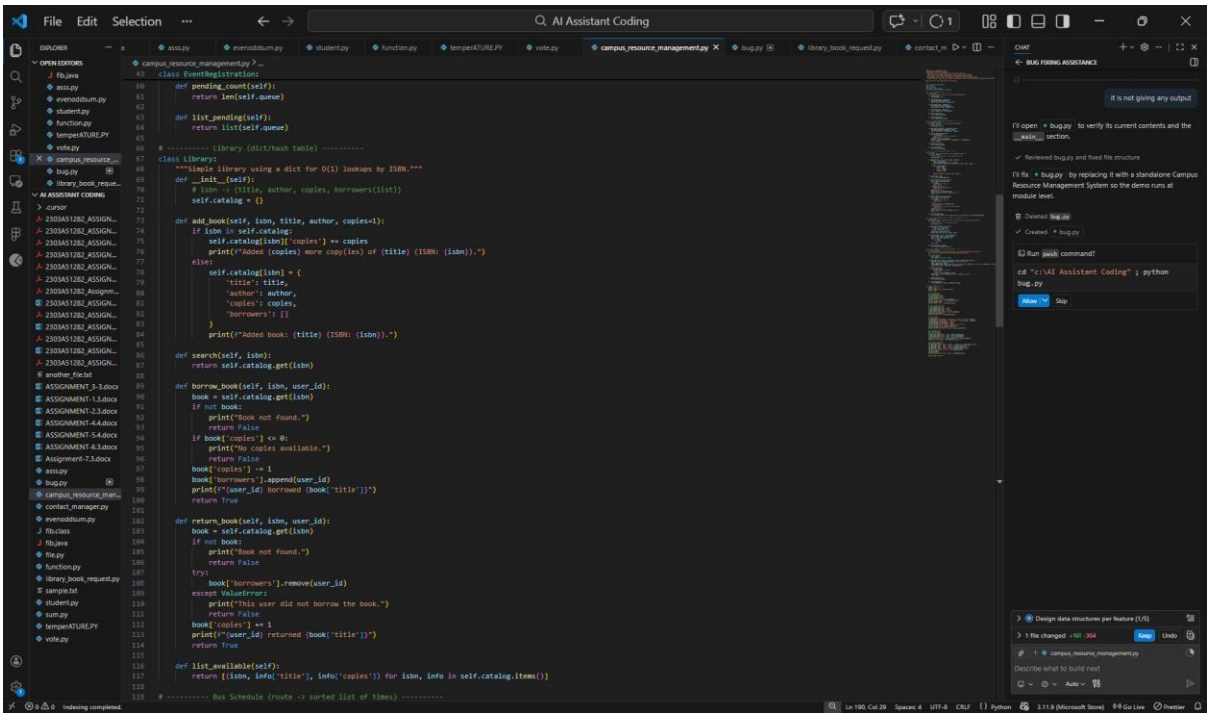
## Prompt:

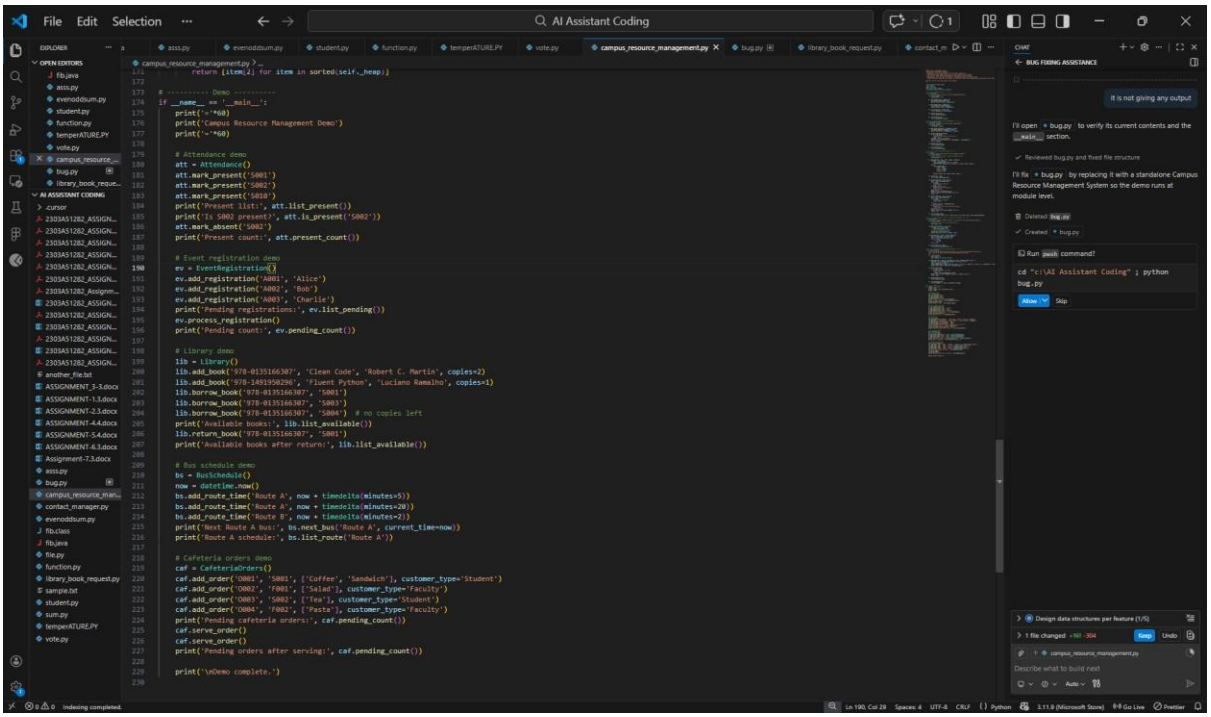
Create a Campus Resource Management System in Python.For each feature (Attendance, Event Registration, Library, Bus Schedule, Cafeteria Orders), choose the best data structure



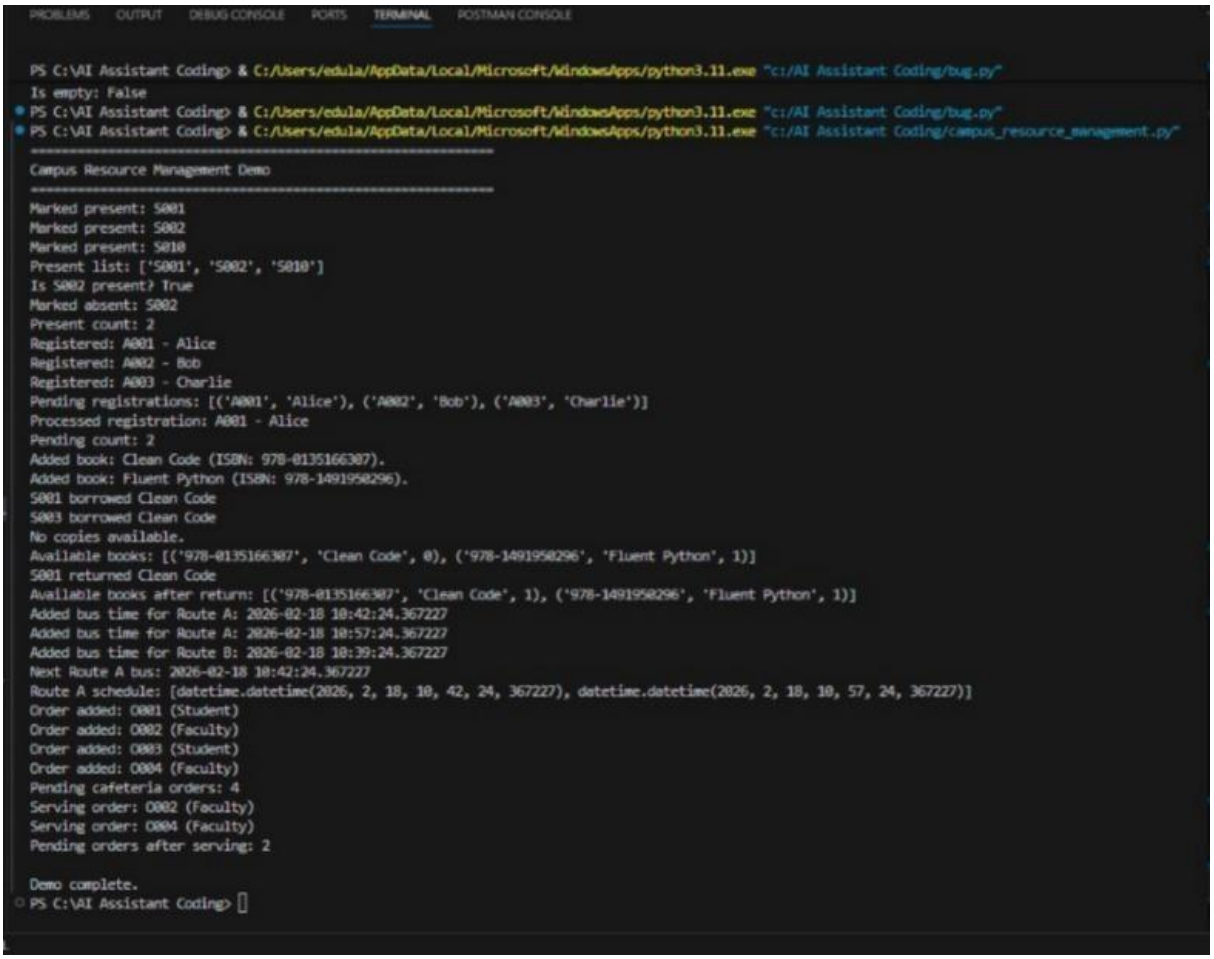

## Code:




File Edit Selection
AI Assistant Coding
class EventRegistration:
def pending_count(self):
return len(self.queue)
def list_pending(self):
return list(self.queue)
# ---------- Library (dict/hash table) ----------
class Library:
"""Simple library using a dict for O(1) lookups by ISBN."""
def __init__(self):
# isbn -> {title, author, copies, borrowers(list)}
self.catalog = {}
def add_book(self, isbn, title, author, copies=1):
if isbn in self.catalog:
self.catalog[isbn]['copies'] += copies
print(f"Added {copies} more copy(ies) of {title} (ISBN: {isbn}).")
else:
self.catalog[isbn] = {
'title': title,
'author': author,
'copies': copies,
'borrowers': []
}
print(f"Added book: {title} (ISBN: {isbn}).")
def search(self, isbn):
return self.catalog.get(isbn)
def borrow_book(self, isbn, user_id):
book = self.catalog.get(isbn)
if not book:
print("Book not found.")
return False
if book['copies'] <= 0:
print("No copies available.")
return False
book['copies'] -= 1
book['borrowers'].append(user_id)
print(f"{user_id} borrowed {book['title']}")
return True
def return_book(self, isbn, user_id):
book = self.catalog.get(isbn)
if not book:
print("Book not found.")
return False
try:
book['borrowers'].remove(user_id)
except ValueError:
print("This user did not borrow the book.")
return False
book['copies'] += 1
print(f"{user_id} returned {book['title']}")
return True
def list_available(self):
return [(isbn, info['title'], info['copies']) for isbn, info in self.catalog.items()]
# ---------- Bus Schedule (route -> sorted list of times) ----------
BUG FIXING ASSISTANCE
it is not giving any output


File Edit Selection
AI Assistant Coding
return [item[1] for item in sorted(self._heap)]
# ---------- Demo ----------
if __name__ == '__main__':
print('='*60)
print('Campus Resource Management Demo')
print('='*60)
# Attendance demo
att = Attendance()
att.mark_present('S001')
att.mark_present('S002')
att.mark_present('S010')
print('Present list:', att.list_present())
print('Is S002 present?', att.is_present('S002'))
att.mark_absent('S002')
print('Present count:', att.present_count())
# Event registration demo
ev = EventRegistration()
ev.add_registration('A001', 'Alice')
ev.add_registration('A002', 'Bob')
ev.add_registration('A003', 'Charlie')
print('Pending registrations:', ev.list_pending())
ev.process_registration()
print('Pending count:', ev.pending_count())
# Library demo
lib = Library()
lib.add_book('978-0135166307', 'Clean Code', 'Robert C. Martin', copies=2)
lib.add_book('978-1491950296', 'Fluent Python', 'Luciano Ramalho', copies=1)
lib.borrow_book('978-0135166307', 'S001')
lib.borrow_book('978-0135166307', 'S003')
lib.borrow_book('978-0135166307', 'S004') # no copies left
print('Available books:', lib.list_available())
lib.return_book('978-0135166307', 'S001')
print('Available books after return:', lib.list_available())
# Bus schedule demo
bs = BusSchedule()
now = datetime.now()
bs.add_route_time('Route A', now + timedelta(minutes=5))
bs.add_route_time('Route A', now + timedelta(minutes=20))
bs.add_route_time('Route B', now + timedelta(minutes=2))
print('Next Route A bus:', bs.next_bus('Route A', current_time=now))
print('Route A schedule:', bs.list_route('Route A'))
# Cafeteria orders demo
caf = CafeteriaOrders()
caf.add_order('O001', 'S003', ['Coffee', 'Sandwich'], customer_type='Student')
caf.add_order('O002', 'F001', ['Salad'], customer_type='Faculty')
caf.add_order('O003', 'S002', ['Tea'], customer_type='Student')
caf.add_order('O004', 'F002', ['Pasta'], customer_type='Faculty')
print('Pending cafeteria orders:', caf.pending_count())
caf.serve_order()
caf.serve_order()
print('Pending orders after serving:', caf.pending_count())
print('\nDemo complete.')
BUG FIXING ASSISTANCE
it is not giving any output

# Output:




**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.