SR UNIVERSITY

School of Computer Science and Artificial Intelligence

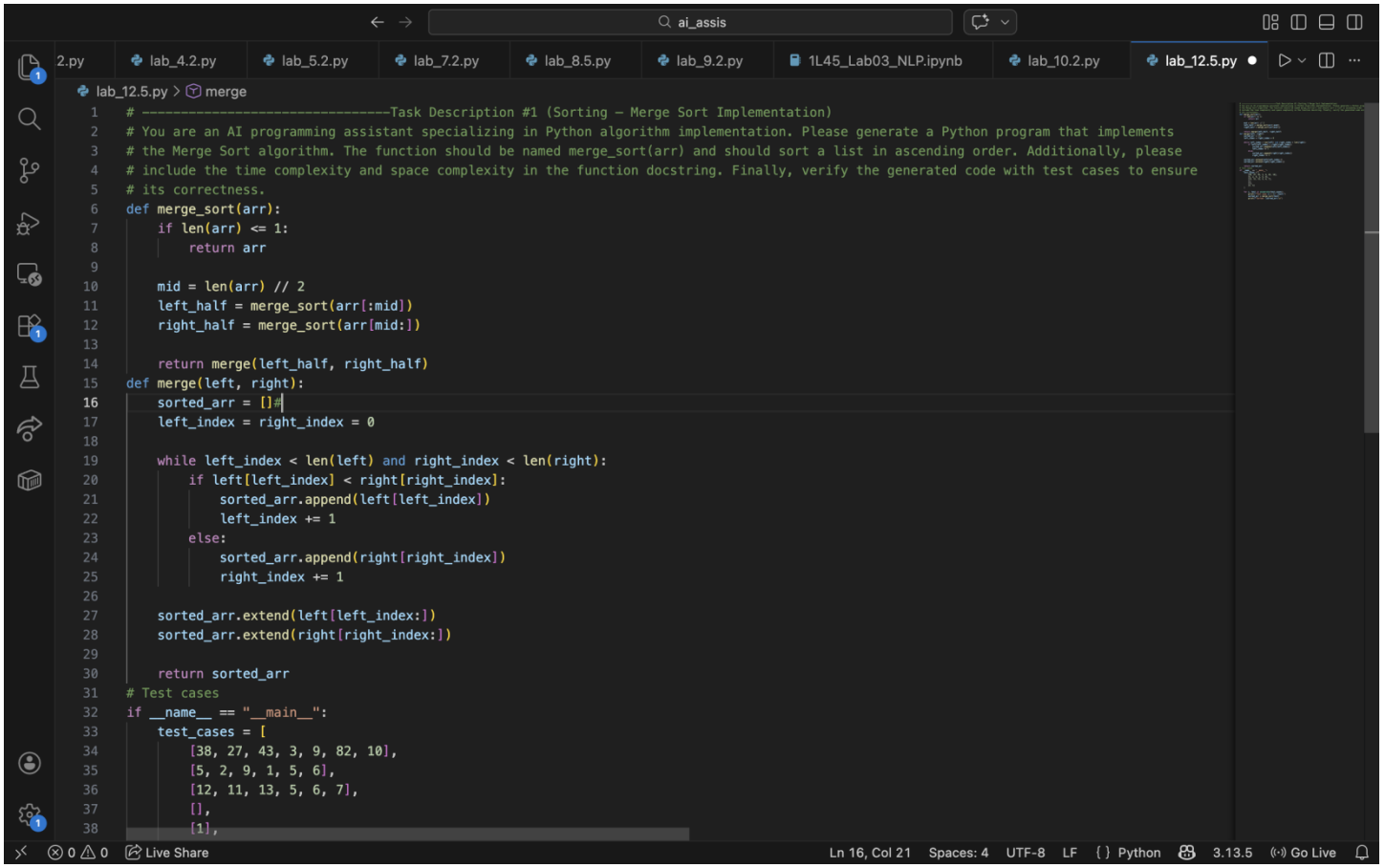
Lab Assignment # 12.5

Program	**: B. Tech (CSE)**
Specialization	**: --**
Course Title	**: AI Assisted coding**
Course Code	**:**
Semester	**II**
Academic Session	**: 2025-2026**
Name of Student	**: Akash Reddy**
Enrollment No.	**: 2403A51L30**
Batch No.	**: 51**
Date	**:20-02-2026**

Task Description #1 (Sorting – Merge Sort Implementation):

Prompt: You are an AI programming assistant specializing in Python algorithm implementation. Please generate a Python program that implementsthe Merge Sort algorithm. The function should be named merge_sort(arr) and should sort a list in ascending order. Additionally, please include the time complexity and space complexity in the function docstring. Finally, verify the generated code with test cases to ensure

Output:



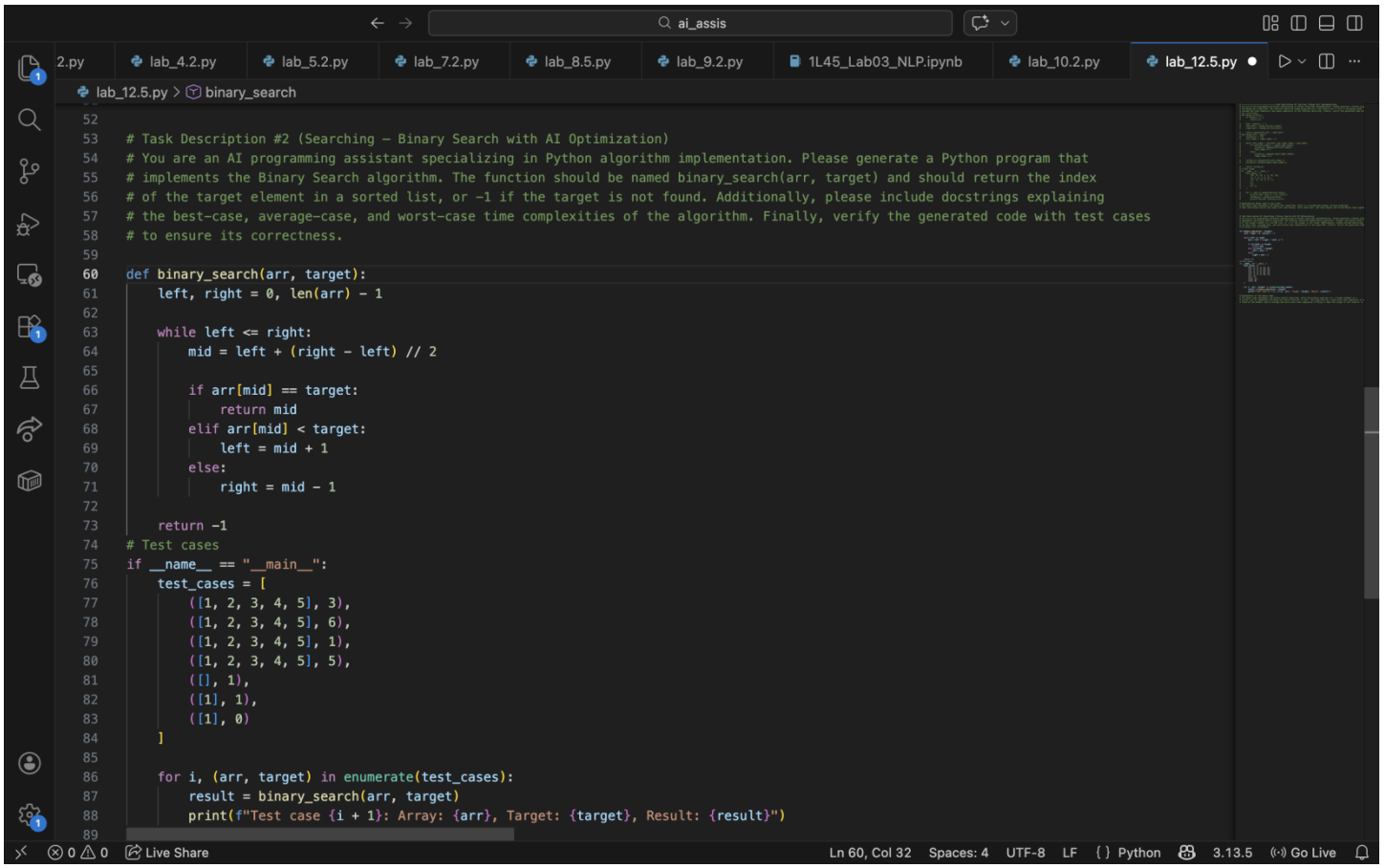
```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Test case 1: [38, 27, 43, 3, 9, 82, 10]
Sorted: [3, 9, 10, 27, 38, 43, 82]

Test case 2: [5, 2, 9, 1, 5, 6]
Sorted: [1, 2, 5, 5, 6, 9]

Test case 3: [12, 11, 13, 5, 6, 7]
Sorted: [5, 6, 7, 11, 12, 13]

Test case 4: []
Sorted: []

Test case 5: [1]
Sorted: [1]

Test case 6: [2, 1]
Sorted: [1, 2]

(base) akash@AKASHs-MacBook-Air ai_assis %
```

Expalanation: The above code implements the Merge Sort algorithm, which is a divide-and-conquer sorting technique that recursively splits the input list into halves, sorts each half, and then merges the sorted halves back together. The time complexity of Merge Sort is O(n log n) and the space complexity is O(n).

Task Description #2 (Searching – Binary Search with AI Optimization):

Prompt: You are an AI programming assistant specializing in Python algorithm implementation. Please generate a Python program thatimplements the Binary Search algorithm. The function should be named binary_search(arr, target) and should return the index of the target element in a sorted list, or -1 if the target is not found. Additionally, please include docstrings explaining the best-case, average-case, and worst-case time complexities of the algorithm. Finally, verify the generated code with test cases

Output:



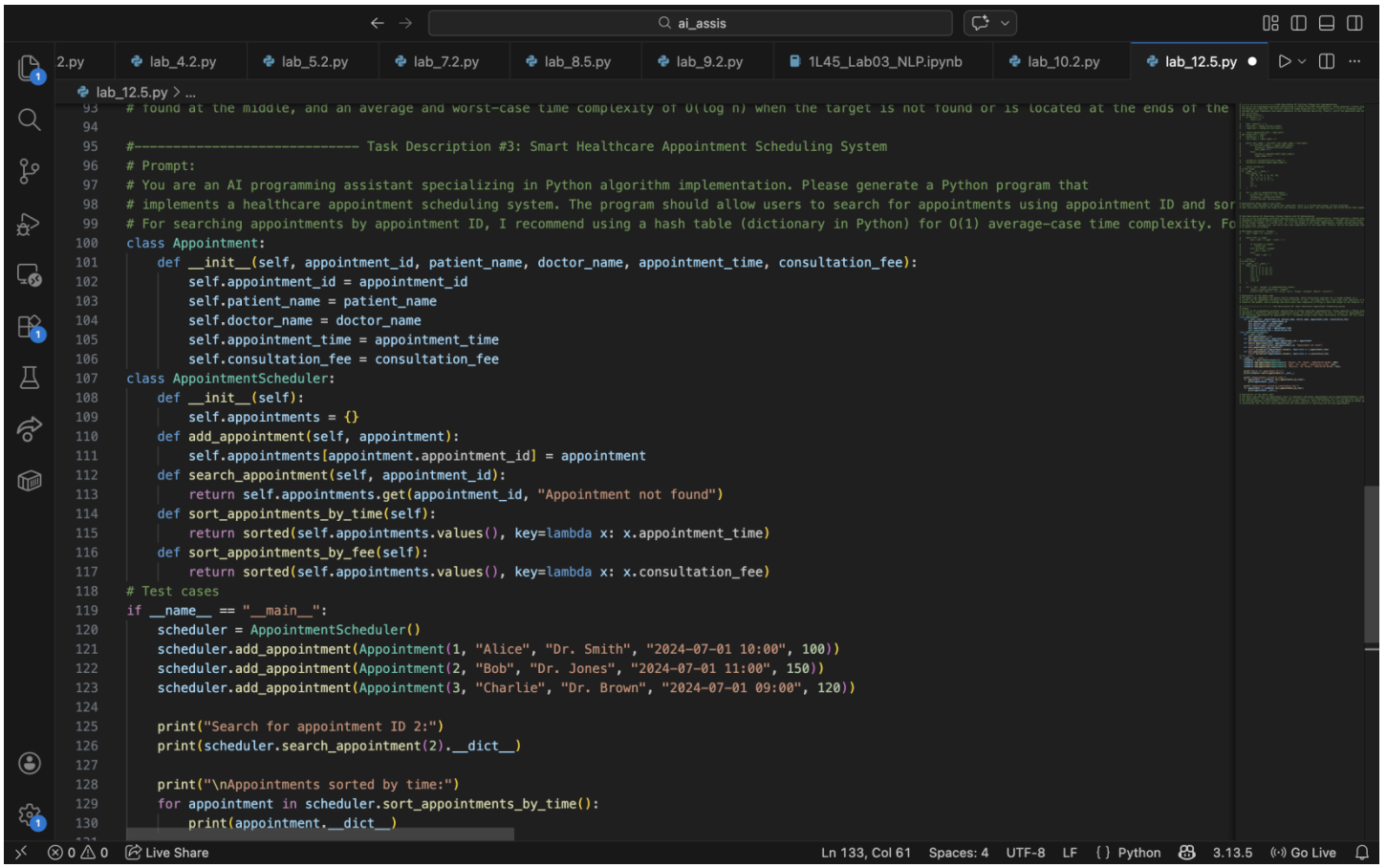
```
88          print(f"Test case {i + 1}: Array: {arr}, Target: {target}, Result: {result}")

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                   Python

/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Test case 1: Array: [1, 2, 3, 4, 5], Target: 3, Result: 2
Test case 2: Array: [1, 2, 3, 4, 5], Target: 6, Result: -1
Test case 3: Array: [1, 2, 3, 4, 5], Target: 1, Result: 0
Test case 4: Array: [1, 2, 3, 4, 5], Target: 5, Result: 4
Test case 5: Array: [], Target: 1, Result: -1
Test case 6: Array: [1], Target: 1, Result: 0
Test case 7: Array: [1], Target: 0, Result: -1
(base) akash@AKASHs-MacBook-Air ai_assis %
```

Explanation: The above code implements the Binary Search algorithm, which efficiently searches for a target element in asorted list by repeatedly dividing the search interval in half. The algorithm has a best-case time complexity of O(1) when the target is found at the middle, and an average and worst-case time complexity of O(log n) when the target is not found or is located at the ends of the list.

Task Description #3: Smart Healthcare Appointment Scheduling System:

Prompt: You are an AI programming assistant specializing in Python algorithm implementation. Please generate a Python program that implements a healthcare appointment scheduling system. The program should allow users to search for appointments using appointment ID and sort appointments based on time or consultation fee. Please recommend suitable searching and sorting algorithms for these tasks, justify your choices, and implement the algorithms in Python. Finally, verify the generated code with test cases to ensure its correctness.

Output:



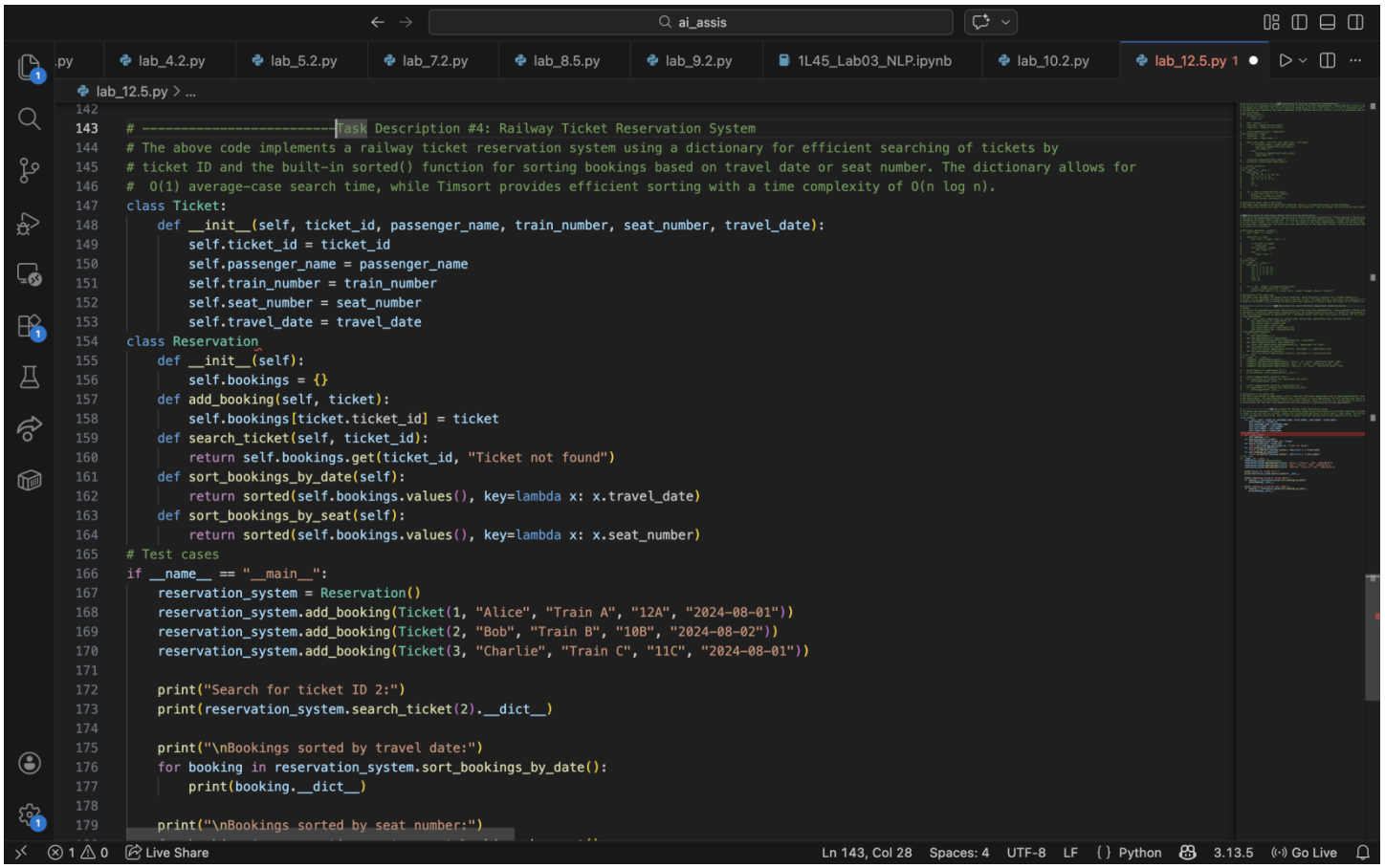
```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Search for appointment ID 2:
{'appointment_id': 2, 'patient_name': 'Bob', 'doctor_name': 'Dr. Jones', 'appointment_time': '2024-07-01 11:00', 'consultation_fee': 150}

Appointments sorted by time:
{'appointment_id': 3, 'patient_name': 'Charlie', 'doctor_name': 'Dr. Brown', 'appointment_time': '2024-07-01 09:00', 'consultation_fee': 120}
{'appointment_id': 1, 'patient_name': 'Alice', 'doctor_name': 'Dr. Smith', 'appointment_time': '2024-07-01 10:00', 'consultation_fee': 100}
{'appointment_id': 2, 'patient_name': 'Bob', 'doctor_name': 'Dr. Jones', 'appointment_time': '2024-07-01 11:00', 'consultation_fee': 150}

Appointments sorted by consultation fee:
{'appointment_id': 1, 'patient_name': 'Alice', 'doctor_name': 'Dr. Smith', 'appointment_time': '2024-07-01 10:00', 'consultation_fee': 100}
{'appointment_id': 3, 'patient_name': 'Charlie', 'doctor_name': 'Dr. Brown', 'appointment_time': '2024-07-01 09:00', 'consultation_fee': 120}
{'appointment_id': 2, 'patient_name': 'Bob', 'doctor_name': 'Dr. Jones', 'appointment_time': '2024-07-01 11:00', 'consultation_fee': 150}
(base) akash@AKASHs-MacBook-Air ai_assis %
```

Explanation: The above code defines an Appointment class to represent individual appointments and an AppointmentScheduler class to manage the appointments. The AppointmentScheduler uses a dictionary to store appointments for O(1) average-case search time by appointment ID. The sorting methods utilize Python's built-in sorted() function, which is efficient for sorting based on either appointment time or consultation fee. The test cases demonstrate the functionality of searching and sorting appointments.

Task Description #4: Railway Ticket Reservation System Scenario:

Prompt: The above code implements a railway ticket reservation system using a dictionary for efficient searching of tickets by ticket ID and the built-in sorted() function for sorting bookings based on travel date or seat number. The dictionary allows for O(1) average-case search time, while Timsort provides efficient sorting with a time complexity of O(n log n).

Output:



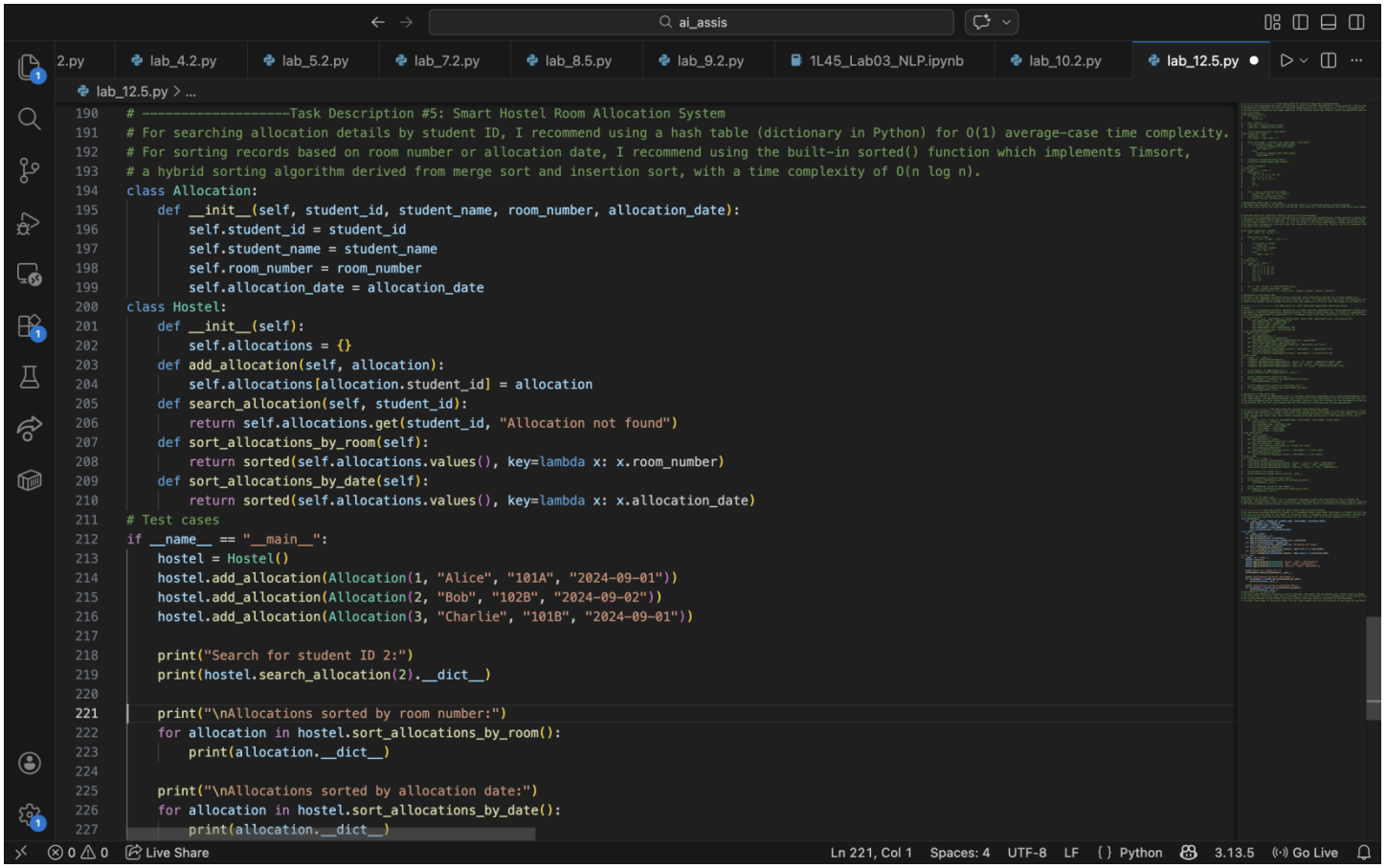
```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Search for ticket ID 2:
{'ticket_id': 2, 'passenger_name': 'Bob', 'train_number': 'Train B', 'seat_number': '10B', 'travel_date': '2024-08-02'}

Bookings sorted by travel date:
{'ticket_id': 1, 'passenger_name': 'Alice', 'train_number': 'Train A', 'seat_number': '12A', 'travel_date': '2024-08-01'}
{'ticket_id': 3, 'passenger_name': 'Charlie', 'train_number': 'Train C', 'seat_number': '11C', 'travel_date': '2024-08-01'}
{'ticket_id': 2, 'passenger_name': 'Bob', 'train_number': 'Train B', 'seat_number': '10B', 'travel_date': '2024-08-02'}

Bookings sorted by seat number:
{'ticket_id': 2, 'passenger_name': 'Bob', 'train_number': 'Train B', 'seat_number': '10B', 'travel_date': '2024-08-02'}
{'ticket_id': 3, 'passenger_name': 'Charlie', 'train_number': 'Train C', 'seat_number': '11C', 'travel_date': '2024-08-01'}
{'ticket_id': 1, 'passenger_name': 'Alice', 'train_number': 'Train A', 'seat_number': '12A', 'travel_date': '2024-08-01'}
(base) akash@AKASHs-MacBook-Air ai_assis %
```

Explanation: The above code defines a Ticket class to represent individual tickets and a Reservation class to manage the bookings. The Reservation class uses a dictionary to store bookings for O(1) average-case search time by ticket ID.The sorting methods utilize Python's built-in sorted() function, which is efficient for sorting based on either travel date or seat number.

Task Description #5:

Prompt: For searching allocation details by student ID, I recommend using a hash table (dictionary in Python) for O(1) average-case time complexity. For sorting records based on room number or allocation date, I recommend using the built-in sorted() function which implements Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of O(n log n).

Output:



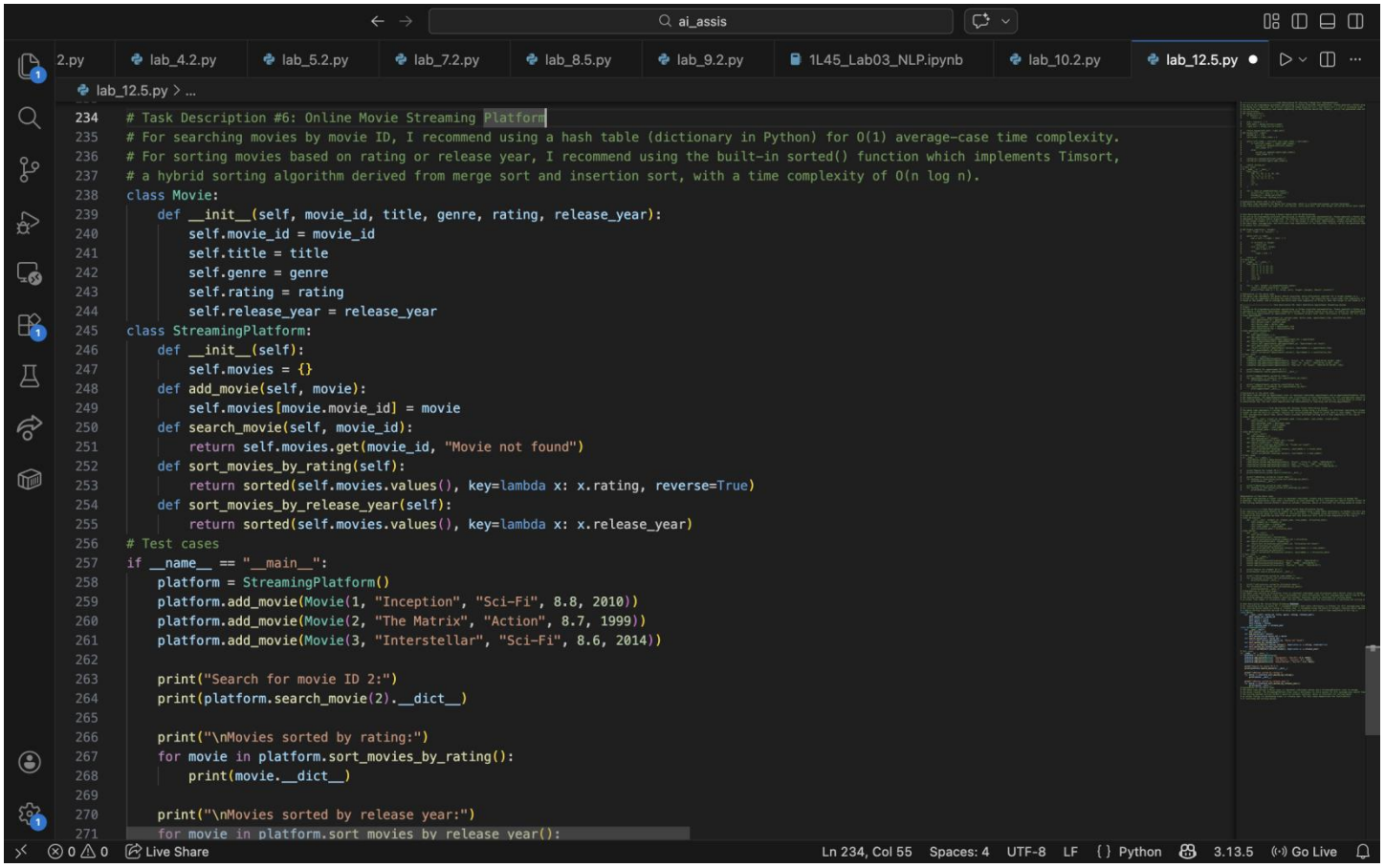
```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Search for student ID 2:
{'student_id': 2, 'student_name': 'Bob', 'room_number': '102B', 'allocation_date': '2024-09-02'}

Allocations sorted by room number:
{'student_id': 1, 'student_name': 'Alice', 'room_number': '101A', 'allocation_date': '2024-09-01'}
{'student_id': 3, 'student_name': 'Charlie', 'room_number': '101B', 'allocation_date': '2024-09-01'}
{'student_id': 2, 'student_name': 'Bob', 'room_number': '102B', 'allocation_date': '2024-09-02'}

Allocations sorted by allocation date:
{'student_id': 1, 'student_name': 'Alice', 'room_number': '101A', 'allocation_date': '2024-09-01'}
{'student_id': 3, 'student_name': 'Charlie', 'room_number': '101B', 'allocation_date': '2024-09-01'}
{'student_id': 2, 'student_name': 'Bob', 'room_number': '102B', 'allocation_date': '2024-09-02'}
(base) akash@AKASHs-MacBook-Air ai_assis %
```

Explanation: The above code defines an Allocation class to represent individual room allocations and a Hostel class to managethe allocations. The Hostel class uses a dictionary to store allocations for O(1) average-case search time by student ID. The sorting methods utilize Python's built-in sorted() function, which is efficient for sorting based on either room number or allocation date. The test cases demonstrate the functionality of searching and sorting allocations.

Task Description #6: Online Movie Streaming Platform:

Prompt: For searching movies by movie ID, I recommend using a hash table (dictionary in Python) for O(1) average-case time complexity.For sorting movies based on rating or release year, I recommend using the built-in sorted() function which implements Timsort,a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of O(n log n).

Output:



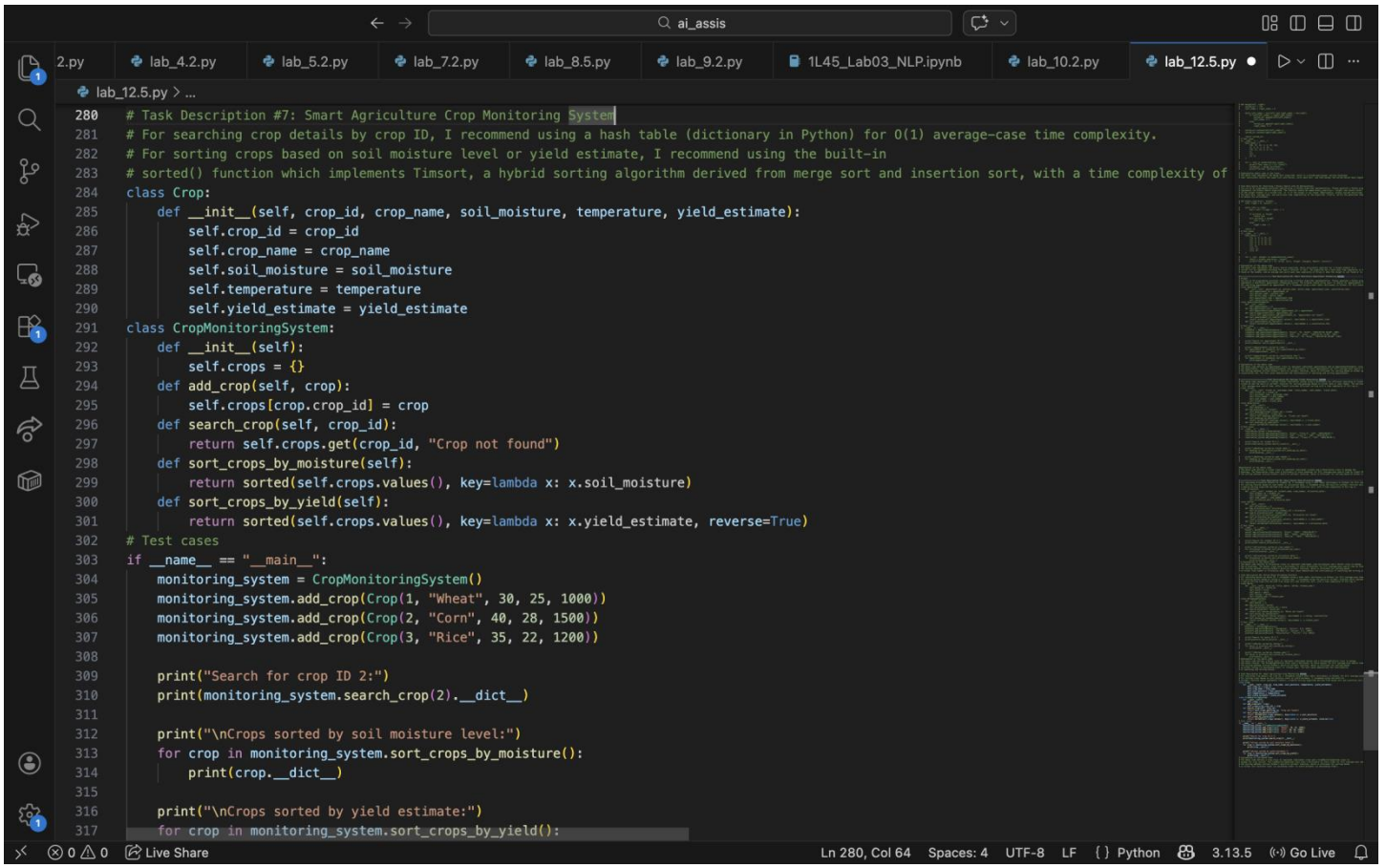
```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Search for movie ID 2:
{'movie_id': 2, 'title': 'The Matrix', 'genre': 'Action', 'rating': 8.7, 'release_year': 1999}

Movies sorted by rating:
{'movie_id': 1, 'title': 'Inception', 'genre': 'Sci-Fi', 'rating': 8.8, 'release_year': 2010}
{'movie_id': 2, 'title': 'The Matrix', 'genre': 'Action', 'rating': 8.7, 'release_year': 1999}
{'movie_id': 3, 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}

Movies sorted by release year:
{'movie_id': 2, 'title': 'The Matrix', 'genre': 'Action', 'rating': 8.7, 'release_year': 1999}
{'movie_id': 1, 'title': 'Inception', 'genre': 'Sci-Fi', 'rating': 8.8, 'release_year': 2010}
{'movie_id': 3, 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}
(base) akash@AKASHs-MacBook-Air ai_assis %
```

Explanation: The above code defines a Movie class to represent individual movies and a StreamingPlatform class to manage the movie records. The StreamingPlatform class uses a dictionary to store movies for O(1) average-case search time by movie ID.The sorting methods utilize Python's built-in sorted() function, which is efficient for sorting basedon either rating (in descending order) or release year. The test cases demonstrate the functionality

Task Description #7: Smart Agriculture Crop Monitoring System:

Prompt: For searching crop details by crop ID, I recommend using a hash table (dictionary in Python) for O(1) average-case time complexity.For sorting crops based on soil moisture level or yield estimate, I recommend using the built-insorted() function which implements Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of O(n log n).

Output:



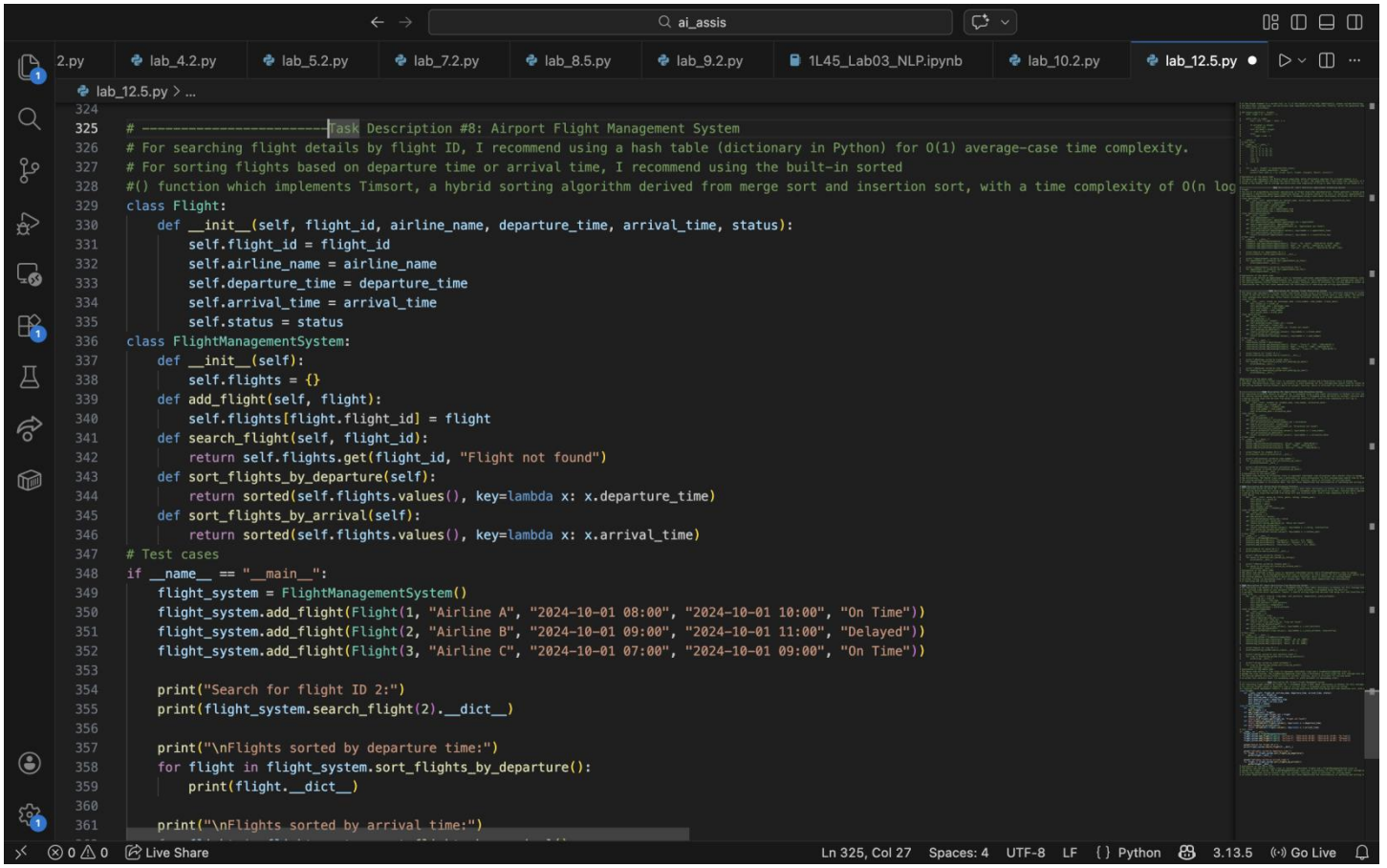
```
/usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
(base) akash@AKASHs-MacBook-Air ai_assis % /usr/local/bin/python3 /Users/akash/Desktop/ai_assis/lab_12.5.py
Search for crop ID 2:
{'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture': 40, 'temperature': 28, 'yield_estimate': 1500}

Crops sorted by soil moisture level:
{'crop_id': 1, 'crop_name': 'Wheat', 'soil_moisture': 30, 'temperature': 25, 'yield_estimate': 1000}
{'crop_id': 3, 'crop_name': 'Rice', 'soil_moisture': 35, 'temperature': 22, 'yield_estimate': 1200}
{'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture': 40, 'temperature': 28, 'yield_estimate': 1500}

Crops sorted by yield estimate:
{'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture': 40, 'temperature': 28, 'yield_estimate': 1500}
{'crop_id': 3, 'crop_name': 'Rice', 'soil_moisture': 35, 'temperature': 22, 'yield_estimate': 1200}
{'crop_id': 1, 'crop_name': 'Wheat', 'soil_moisture': 30, 'temperature': 25, 'yield_estimate': 1000}
(base) akash@AKASHs-MacBook-Air ai_assis %
```

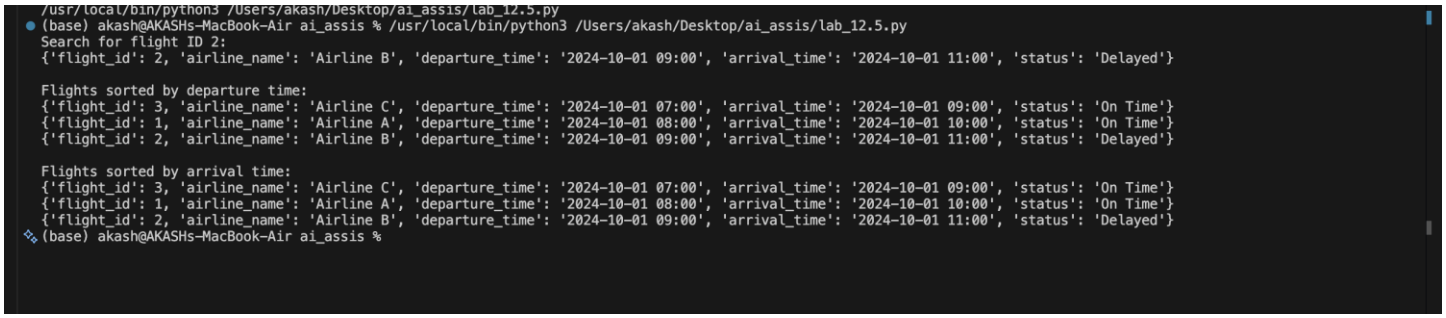
Explanation: The above code defines a Crop class to represent individual crops and a CropMonitoringSystem class to manage the crop records. The CropMonitoringSystem class uses a dictionary to store crops for O(1) average-case search time by crop ID.The sorting methods utilize Python's built-in sorted() function, which is efficient for sorting based on either soil moisture level (in ascending order) or yield estimate (in descending order).

Task Description #8: Airport Flight Management System:

Prompt: For searching flight details by flight ID, I recommend using a hash table (dictionary in Python) for O(1) average-case time complexity. For sorting flights based on departure time or arrival time, I recommend using the built-in sorted() function which implements Timsort, a hybrid sorting algorithm derived from merge sort and insertion sort, with a time complexity of O(n log n).

Output:





Explanation: The above code defines a Flight class to represent individual flights and a FlightManagementSystem class tomanage the flight records. The FlightManagementSystem class uses a dictionary to store flights for O(1) average-case search time by flight ID.The sorting methods utilize Python's built-in sorted() function, which is efficient for sorting basedon either departure time or arrival time. The test cases demonstrate the functionality of searching and sorting flights.