Week 3: Industrial Training Report

Overview

The third week of industrial training at **Auribises Technologies Pvt. Ltd.** focused on data structures, object relationships, and search and sort algorithms in Python. The aim was to build an understanding of how to store, retrieve, and manage data efficiently using different approaches like lists, dictionaries, and objects. Additionally, advanced algorithms such as binary search, bubble sort, merge sort, and quick sort were implemented and analyzed.

Day 11: Understanding Classes and Has-a Relationship

We began the week by understanding how **object relationships** work in Python, specifically the **has-a relationship**. This concept defines how one class can contain another as an attribute. It forms the basis of real-world modeling.

We also started implementing data structures like **lists and dictionaries** to store and organize structured data.

Topics Covered:

- Class and Object Implementation
- Has-a Relationship in Python
- Creating Dictionaries for Real-World Entities

Example: Has-a Relationship with Flight and Airline Classes

```
class Airline:
    def __init__(self, name):
        self.name = name

class Flight:
    def __init__(self, flight_no, airline):
        self.flight_no = flight_no
        self.airline = airline  # Has-a relationship

    def show_details(self):
        print(f"Flight: {self.flight_no}, Airline: {self.airline.name}")
```

```
air_india = Airline("Air India")
flight1 = Flight("AI302", air_india)
flight1.show_details()
```

This example shows that a flight *has an* airline, establishing a relationship between two classes.

Day 12: Search Operations in Python

The twelfth day was dedicated to implementing **search operations** on different data structures. We started with lists, then moved to dictionaries and lists of objects. We explored both **linear search** and **binary search**, comparing their efficiency and time complexity.

Topics Covered:

- Linear and Binary Search Concepts
- Searching Data in Lists, Dictionaries, and Objects
- Time Complexity Comparison

Example 1: Linear Search in a List

```
def linear_search(data, key):
    for i in range(len(data)):
        if data[i] == key:
            return i
    return -1

nums = [10, 20, 30, 40, 50]
key = 30
index = linear_search(nums, key)
print(f"Element found at index: {index}")
```

Example 2: Binary Search in a Sorted List

```
def binary_search(data, key):
    low = 0
    high = len(data) - 1
    while low <= high:
        mid = (low + high) // 2
        if data[mid] == key:
```

```
            return mid
        elif data[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1

sorted_nums = [10, 20, 30, 40, 50, 60]
print(binary_search(sorted_nums, 40))
```

Binary search divides the list into halves, making it faster than linear search for sorted data.

Day 13: Sorting Techniques and Filtering Data

Day thirteen focused on **sorting algorithms**. Sorting organizes data in a specific order—ascending or descending—and is fundamental in data processing. We implemented **bubble sort**, **merge sort**, and **quick sort**. Additionally, we practiced **filtering** data based on multiple conditions.

Topics Covered:

- Bubble, Merge, and Quick Sort Algorithms
- Sorting Strings and Numbers
- Filtering Based on Multiple Attributes

Example 1: Bubble Sort

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
print("Sorted Array:", arr)
```

Example 2: Merge Sort

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

arr = [38, 27, 43, 3, 9, 82, 10]
merge_sort(arr)
print("Sorted Array:", arr)
```

Example 3: Filtering Flights Using Dictionaries

```
flights = [
    {"code": "AI101", "price": 4500, "destination": "Mumbai"},
    {"code": "AI202", "price": 7000, "destination": "Bangalore"},
    {"code": "AI303", "price": 3500, "destination": "Delhi"}
]

cheap_flights = [f for f in flights if f["price"] < 5000]
print("Flights below Rs.5000:", cheap_flights)
```

Filtering allows multiple conditions, unlike sorting which happens on a single attribute.

Day 14: Linked List and Stack Implementation

The fourteenth day was dedicated to implementing **linked lists** and **stacks**, two essential data structures that handle dynamic data efficiently. We understood how to create a linked list using Python classes and performed basic stack operations like push and pop.

Topics Covered:

- Linked List Node Creation
- Stack Operations Using Lists
- Understanding Dynamic Memory Allocation

Example 1: Linked List Implementation

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    def display(self):
        temp = self.head
        while temp:
```

```
        print(temp.data, end=" -> ")
        temp = temp.next

ll = LinkedList()
ll.append(10)
ll.append(20)
ll.append(30)
ll.display()
```

Example 2: Stack Operations

```
stack = []
stack.append(10)
stack.append(20)
stack.append(30)
print("Stack:", stack)
stack.pop()
print("After Pop:", stack)
```

Day 15: Combined Operations and Application Practice

The final day of the week involved combining all learned concepts — data structures, searching, sorting, and filtering — into one complete program. We built a small **Flight Management System** using lists, dictionaries, and classes.

Topics Covered:

- Combined Search, Sort, and Filter
- Data Structure Integration
- Practical Implementation using Class-based Design

Example: Flight Management System

```
class Flight:
    def __init__(self, code, destination, price):
        self.code = code
        self.destination = destination
        self.price = price
```

```
flights = [
    Flight("AI101", "Mumbai", 4500),
    Flight("AI202", "Delhi", 5200),
    Flight("AI303", "Bangalore", 7000)
]

# Search Operation
for f in flights:
    if f.destination == "Delhi":
        print(f"Found: {f.code} to {f.destination}")

# Sort Operation (Low to High)
flights.sort(key=lambda x: x.price)
print("Flights sorted by price:")
for f in flights:
    print(f"{f.code} - Rs.{f.price}")
```

Summary

Week 3 focused on applying data structure concepts in Python, emphasizing efficient data handling through searching, sorting, and filtering mechanisms. Implementations of linked lists and stacks gave practical exposure to dynamic memory management. The week concluded with the development of a mini **Flight Management System**, consolidating all the operations learned so far.