Case Study 1: Student Marks Management System

Scenario Description:

A school maintains students' Mathematics marks using a Python-based digital record system. The marks are stored in a list so that teachers can easily update, analyze, and manage the class performance data.
Throughout the semester, new students join, some transfer out, and others improve their performance — all of which are dynamically reflected in the system.

Sample Input Data:

Initial marks of students in Class A:
`[85, 90, 78, 92, 88, 76, 95, 89]`

Scenario Flow:

- **New Admission:**
  Midway through the semester, a new student joins and scores **82** in the Mathematics test.
  The mark is added to the existing records.
- **Merging Data:**
  Another section (Class B) is merged with Class A for a combined performance analysis.
  Their marks — `[80, 91, 87]` — are integrated into the same list for consolidated evaluation.
- **Transfer Case:**
  A student who scored **76** transfers to another school.
  Their record is removed from the system to keep data accurate.
- **Error Correction:**
  During data entry, one duplicate score was mistakenly added.
  The last entry is removed to maintain clean data.
- **Performance Review:**
  The teacher analyzes the data to find:
  - The **highest mark** to identify the top performer.
  - The **lowest mark** to plan remedial sessions.
  - The **total and average marks** for overall class evaluation.
  - The **count of students** after updates.
  - The **position** of a specific score to locate individual records.
  - The **distribution of scores** to understand grade patterns.

Banking Case Study 2: Customer Account Management

Scenario / Description:

A bank wants to manage its **customers' accounts** using a digital system. Each customer has:

- **Account Number** (unique key)
- **Name**
- **Account Type** (Savings / Current)
- **Balance**

Sample Input (Dictionary Representation)

Account Number	Name	Account Type	Balance
101	Alice	Savings	5000
102	Bob	Current	12000
103	Charlie	Savings	7000
104	Diana	Current	15000

Operations / Actions:

1. Deposit `2000` to Account `101`.
2. Withdraw `5000` from Account `102`.
3. Add a new account: `105`, Name: `Eve`, Type: `Savings`, Balance: `8000`.
4. Close Account `103`.
5. List all customer accounts.

101: {"name": "Alice", "type": "Savings", "balance": 5000},

102: {"name": "Bob", "type": "Current", "balance": 12000}

Case Study 3: Online Store Product Tags Management

Description:

An e-commerce platform wants to manage **product tags** for its items. Tags help in **searching and filtering products**, and **duplicate tags** should be avoided.

The system should be able to:

1. Add new tags to a product.
2. Remove obsolete tags.
3. Check if a product has a specific tag.
4. Find all unique tags across multiple products.
5. Identify common tags shared by two products.

Using **sets** is ideal because they **automatically prevent duplicate tags** and allow operations like **union, intersection, and membership checks** efficiently.

Sample Input

Product 1 Tags:

```
{"Electronics", "Laptop", "Gaming", "NewArrival"}
```

Product 2 Tags:

```
{"Laptop", "Office", "Electronics", "Discount"}
```

Operations to Perform:

1. Add a new tag `"Portable"` to Product 1.
2. Remove the tag `"NewArrival"` from Product 1.
3. Check if `"Gaming"` is a tag for Product 1.
4. Find all unique tags across Product 1 and Product 2 (Union).
5. Find tags common to both products (Intersection).

Case Study 4: E-Commerce Product Management System

Description:

An online store wants to manage its products efficiently. Each product has:

- **Product ID** (unique)
- **Name**
- **Category**
- **Price**
- **Tags** (like “Electronics”, “Portable”, “NewArrival”)
- **Ratings** (multiple customer ratings)

Sample Input

List of Products (Sequential Access)

```
products = [
    {"id": 101, "name": "Laptop", "category": "Electronics",
"price": 70000, "tags": {"Electronics", "Portable"},
"ratings": (5, 4, 5)},
    {"id": 102, "name": "Smartphone", "category":
"Electronics", "price": 40000, "tags": {"Electronics",
"Mobile", "Portable"}, "ratings": (4, 5, 4, 5)},
    {"id": 103, "name": "Office Chair", "category":
"Furniture", "price": 8000, "tags": {"Furniture", "Office"},
"ratings": (4, 4, 3)},
]
```

Operations to Perform

1. **Add a new product:**
    - ID: 104, Name: "Headphones", Category: "Electronics", Price: 3500, Tags: {"Electronics", "Audio"}, Ratings: (5, 4)
2. **Update price of product 102** to 42000.
3. **Add a new tag "Discount"** to product 101.
4. **Calculate average rating** for each product.
5. **List all unique tags** across all products.
6. **List all product categories** (without duplicates).

Case Study 5: College Course Registration System

Description

A college wants to manage students and the courses they register for.
Each student has a name, ID, registered courses, and GPA stored in different data structures.

Sample Input

```
students = [
    {"id": 1, "name": "Ravi", "courses": ["Math", "Science",
"English"], "gpa": (8.5, 9.0, 8.8)},
    {"id": 2, "name": "Sneha", "courses": ["Math",
"History"], "gpa": (9.2, 9.0)},
    {"id": 3, "name": "Karan", "courses": ["Science",
"Computer"], "gpa": (7.5, 8.0)}
]
```

Operations to Perform

1. Add a new student: ID 4, Name "Priya", Courses `["Math", "Computer", "Economics"]`, GPA `(8.8, 9.1, 9.0)` using `append()`.
2. Add a new course "AI" to Ravi's courses using `insert()`.
3. Remove course "History" from Sneha using `remove()`.
4. Find average GPA for each student using `sum()` and `len()`.
5. Find all unique courses offered using `set()` and `update()`.
6. Sort students alphabetically by name using `sort()`.

Expected Output

```
Added new student: Priya
Updated courses for Ravi: ['Math', 'AI', 'Science',
'English']
Removed 'History' from Sneha
Average GPA:
Ravi: 8.77 | Sneha: 9.1 | Karan: 7.75 | Priya: 8.97
All unique courses: {'Computer', 'Math', 'Science',
'English', 'Economics', 'AI'}
Students sorted by name: ['Karan', 'Priya', 'Ravi', 'Sneha']
```

Case Study 6: Retail Store Sales Tracker

Description

A retail store keeps track of daily sales, product categories, and discounts.
Each day's data is stored in dictionaries inside a list for easy management.

Sample Input

```
sales_data = [
    {"day": "Monday", "sales": [1500, 2300, 1800],
"categories": {"Electronics", "Groceries"}},
    {"day": "Tuesday", "sales": [2500, 1700, 2900],
"categories": {"Clothing", "Groceries"}},
    {"day": "Wednesday", "sales": [3100, 2200, 3300],
"categories": {"Electronics", "Furniture"}}
]
```

Operations to Perform

1. Add new day data for Thursday using `append()`.
2. Remove "Tuesday" data using `pop()` and store it separately.
3. Calculate total and average sales for each day using `sum()` and `len()`.
4. Find all categories sold during the week using `union()` of sets.
5. Add a new category "Discounted" to all days using `add()`.
6. Sort days by total sales using `sorted()` and a lambda function.

Expected Output

```
Added new day: Thursday
Removed Tuesday data
Total & Average Sales:
Monday: 5600 (avg: 1866.67)
Wednesday: 8600 (avg: 2866.67)
Thursday: 9000 (avg: 3000.0)
All categories this week: {'Furniture', 'Discounted',
'Electronics', 'Groceries'}
Sorted days by total sales: [('Monday', 5600), ('Wednesday',
8600), ('Thursday', 9000)]
```

Case Study 7: Banking Transaction Analysis

Description

A bank monitors customer accounts and transaction types to find high-value customers and frequently used services.
Each customer's details are stored in a dictionary inside a list.

Sample Input

```
accounts = [
    {"id": 1001, "name": "Asha", "balance": 95000,
"transactions": [10000, -2000, 3000], "services": {"Loan",
"Insurance"}},
    {"id": 1002, "name": "Rahul", "balance": 45000,
"transactions": [5000, -1000, 1500], "services": {"Credit
Card"}},
    {"id": 1003, "name": "Nisha", "balance": 120000,
"transactions": [20000, -5000, 25000], "services": {"Loan",
"Credit Card"}}
]
```

Operations to Perform

1. Add a new account for "Vikram" with balance 80000, transactions `[10000, -2000, 5000]`, services `{"Insurance"}` using `append()`.
2. Increase Rahul's balance by ₹10,000 using `update()`.
3. Identify VIP customers (balance > ₹80,000) using list comprehension and `filter()`.
4. Calculate average transaction amount for each customer.
5. Find customers who have both "Loan" and "Credit Card" using `intersection()`.
6. Delete Asha's record using `remove()`.

Expected Output

```
Added new account: Vikram
Rahul's updated balance: 55000
VIP Customers: ['Asha', 'Nisha']
Average Transaction Amounts:
Asha: 3666.67 | Rahul: 1833.33 | Nisha: 13333.33 | Vikram:
4333.33
Customers with both Loan & Credit Card: {'Nisha'}
Removed customer: Asha
```