1. OOPS: Encapsulation (Data Hiding)

Beyond the Basics

Encapsulation is not just "putting things in a class." It is the practice of **Information Hiding**. By making attributes private, we hide the internal representation of an object from the outside.

- **Access Modifiers:**
  - **Public:** Accessible from anywhere.
  - **Private (`__` or `private`):** Accessible only within the class.
  - **Protected (`_` or `protected`):** Accessible within the class and its subclasses.

Why it improves Maintainability:

If you decide to change the data type of a variable (e.g., changing `age` from an integer to a date), you only need to update the code inside the class. The external code that calls `get_age()` remains exactly the same.

2. OOPS Scenario: Protecting Sensitive Fields

Problem: Why can't we just let a user update their `account_balance` directly?

Analysis:

1. **Validation Logic:** If a field is public, anyone can set `balance = -5000` . By using a **Setter** method (`set_balance(amount)`), we can add logic: `if amount >= 0: self.__balance = amount` .
2. **Read-Only Access:** We might want a user to *see* their password hash but never *change* it directly. We provide a **Getter** but no **Setter**.
3. **Audit Trails:** Getters and Setters allow us to log *who* accessed or changed the data and *when*.

3. Programming: Sliding Window Technique

This technique is used to perform operations on a specific "window" (subset) of data that slides across a larger dataset.

Problem: Find the maximum sum of a subarray of fixed size k.

Approach 1: Brute Force

- Check every possible subarray of size k and sum them up.
- **Complexity:** $O(n \times k)$.

Approach 2: Sliding Window (Optimized)

1. Calculate the sum of the first k elements.

2. "Slide" the window by one position:
   - **Add** the next element in the array.
   - **Subtract** the first element of the *previous* window.
3. Keep track of the maximum sum found during the slide.

- **Complexity:** $O(n)$ because we only visit each element once.

> **Visual Logic:** Instead of re-summing k elements, you just adjust the "edges" of your sum.

4. SQL: CROSS JOIN

A `CROSS JOIN` produces the **Cartesian Product** of two tables.

The Logic: "Every Possible Pair"

If Table A has 10 rows and Table B has 5 rows, a CROSS JOIN results in $10 \times 5 = 50$ rows. It does **not** require an `ON` clause.

Syntax

```
SELECT products.name, colors.color_name
FROM products
CROSS JOIN colors;
```

When to use (and avoid):

- **Use Case:** Generating all possible combinations (e.g., every product in every available color and size).
- **Danger Zone:** Avoid on large tables. Joining a table of 1,000 rows with another of 1,000 rows creates **1,000,000 rows**, which can crash a system or slow down reports significantly.

Summary Table

Topic	Focus	Key Takeaway
OOPS	Getters/Setters	Provides controlled access and validation for sensitive data.
DSA	Sliding Window	Optimized for "Range" problems, reducing $O(n \times k)$ to $O(n)$.
SQL	CROSS JOIN	Creates a Cartesian product; powerful for combinations but risky for performance.

Five days in—your mental model of software architecture is becoming much sharper!