1. OOPS: 'self' & Object Context

What is 'self'?

In Python (and similar concepts like `this` in Java/C++), `self` is a reference to the **current instance** of the class. It is the bridge between the class definition and the specific object's data.

- **Instance Variables:** Variables prefixed with `self.` belong to the object (e.g., `self.name`).
- **Local Variables:** Variables without `self.` exist only inside the method and are destroyed once the method finishes.

Why is it needed?

1. **Differentiation:** It helps the computer distinguish between an instance attribute and a local parameter if they have the same name.
2. **Accessing Methods:** To call one method from another within the same class, you must use `self.method_name()`.

2. OOPS Scenario: The "Who am I?" Problem

Problem: If you have 100 `User` objects and they all call the `display_profile()` method simultaneously, how does the method know which user's data to show?

Analysis:

1. **The Hidden Argument:** When you call `user_a.display_profile()`, Python automatically passes `user_a` as the first argument (`self`) to the method.
2. **Context Maintenance:** Inside the method, `self.name` looks at the memory address of `user_a` specifically.
3. **Conclusion:** Without `self`, methods would be static and unable to interact with the unique data of the individual objects that called them.

3. Programming: Two Pointer Technique

This technique is a staple for optimizing array problems, often reducing complexity from $O(n^2)$ to $O(n)$.

Common Use Case: Reversing an Array or Finding a Target Sum in a Sorted Array.

Implementation (Example: Reverse an Array)

1. Place `pointer_1` at index `0`.
2. Place `pointer_2` at index `length - 1`.
3. While `pointer_1 < pointer_2`:

- Swap elements at these two indices.
- Move `pointer_1` forward (`+1`).
- Move `pointer_2` backward (`-1`).

Why use it?

- **Space Efficiency:** Usually operates "in-place" ($O(1)$ extra space).
- **Time Efficiency:** It ensures we process each element at most once.

4. SQL: FULL OUTER JOIN

The `FULL OUTER JOIN` combines the results of both `LEFT JOIN` and `RIGHT JOIN` .

The Logic: "The Total Intersection & Union"

It returns all records when there is a match in either left or right table records. If there is no match, the missing side contains `NULL` .

Syntax

```
SELECT students.name, courses.course_name
FROM students
FULL OUTER JOIN courses ON students.course_id = courses.course_id;
```

Practical Use Cases:

- **Complete Audit:** Finding all students (even those not enrolled) **AND** all courses (even those with no students).
- **Data Synchronization:** Comparing two tables to find discrepancies on both sides.

Summary Table

Topic	Focus	Key Takeaway
OOPS	`self` Keyword	Provides the execution context; links the method to a specific object's memory.
DSA	Two Pointers	Optimized traversal by moving from both ends (or different speeds) simultaneously.
SQL	FULL OUTER JOIN	The most inclusive join; shows matches and non-matches from both tables.

Four days down, 56 to go! You are mastering the core of system design and logic.