MODULE 1: CORE PYTHON & DATA

WEEK: 3 LECTURE: 14

DATE: 05/09/2025

INSTRUCTOR: HAMZA SAJID

TODAY'S AGENDA

- The "Why" - The Problem with Separate Data
  - Analogy: The Warehouse Manager's Dilemma
- Core Concepts of Joining Tables
  - Understanding Table Aliases (e.g., Products P)
  - The INNER JOIN: Finding Perfect Matches
  - The LEFT JOIN: Finding What's Missing
  - The RIGHT JOIN: The Opposite View
  - The FULL JOIN: The Complete Audit
- Exploring Aggregation
  - The GROUP BY & HAVING clauses
- Putting It All Together & The Hands-On Lab

• THE "WHY" - THE PROBLEM WITH SEPARATE DATA

- Recap: Our store's data is correctly organized in tables like Products and Suppliers. This is efficient for storage.
- The Problem: How do we answer a simple business question like, "Show me each product's name and the name of the company that supplies it?" The data lives in two different places.
- The Inefficient Approach: You could get a list of all products. Then, for each product, you'd have to take its SupplierID and run a new query on the Suppliers table to find the name.
- Analogy: The Warehouse Manager's Dilemma: Imagine you have a clipboard with a list of all Products and their SupplierID. You have a separate, massive binder with the details of all your Suppliers. To find out who to call to reorder "Chai Tea," you first find it on your clipboard, note the SupplierID, then flip through the entire binder to find the supplier with that ID. This is incredibly slow and prone to error.

• THE RELATIONAL SOLUTION: SQL JOINS

- A JOIN is the powerful and efficient solution. It's a single command that tells the database to do the hard work of matching records from the Products and Suppliers tables for you.
- Analogy Continued: A JOIN is like having a modern inventory management system. You type "Chai Tea" into the computer, and it instantly shows you the product details and the full name, address, and phone number of the supplier on a single screen. The system does the lookup for you.
- JOINs are faster, cleaner, and safer than running multiple separate queries.

• BASIC SQL JOINS: "INNER JOIN"

- What it does: The INNER JOIN is the most common type. It returns only the rows where the key exists in both tables. It finds the clean intersection of your data.
- Analogy: The Stockable Products List: This join gives you a list of all products that have a valid, registered supplier. If a product is in your system but its SupplierID is invalid, it won't show up. It's your definitive list of "currently orderable items.“
- **The Query:**
  - SELECT P.ProductName, S.SupplierName FROM Products P INNER JOIN Suppliers S ON P.SupplierID = S.SupplierID;
- This query reads as:
  - "From the Products table, which we'll call P for short, INNER JOIN the Suppliers table, which we'll call S, where the SupplierID in both tables match. For each match, show me the ProductName from table P and the SupplierName from table S."
- A Note on 'P' and 'S' (Aliases):
  - Giving tables a short nickname (an "alias") like P and S is a best practice. It makes your query shorter and easier to read. It is also required if you join two tables that have columns with the exact same name (e.g., Orders.ID and Customers.ID).

BASIC SQL JOINS: "LEFT JOIN"

- What it does: A LEFT JOIN returns all rows from the left table (the first one mentioned), and only the matched rows from the right table. If there is no match, the result is NULL on the right side.
- Analogy: Reviewing Your Supplier Contracts: You want to see your entire Suppliers list to find out if any are inactive. This join gives you a list of every single supplier. Next to their name, it will show the products they supply, or a blank space (NULL) if they supply nothing.
- The Query:
  - SELECT S.SupplierName, P.ProductName FROM Suppliers S LEFT JOIN Products P ON S.SupplierID = P.SupplierID;
- This query reads as:
  - "From the Suppliers table (S), LEFT JOIN the Products table (P) where the IDs match. Show me every SupplierName, and for each one, show the ProductName if a match is found."

• BASIC SQL JOINS: "RIGHT JOIN"

- What it does: A RIGHT JOIN is the mirror opposite of a LEFT JOIN. It returns all rows from the right table (the second one mentioned), and only the matched rows from the left table.
- Analogy: Product Data Integrity Check: You want to see your entire Products list to find any items that are missing a supplier in the system. This join gives you a list of every single product. Next to its name, it will show the supplier's name, or a blank space (NULL) if the SupplierID is invalid or missing.
- The Query:
  - SELECT P.ProductName, S.SupplierName FROM Suppliers S RIGHT JOIN Products P ON S.SupplierID = P.SupplierID;
- This query reads as:
  - "From the Suppliers table (S), RIGHT JOIN the Products table (P) where the IDs match. Show me every ProductName, and for each one, show the SupplierName if a match is found."

• A NOTE ON SQLITE: SIMULATING A RIGHT JOIN

- The Problem: The online tool we are using, SQLite, does not support the RIGHT JOIN command. If you run the query from the last slide, you will get an error.
- The Solution: We can achieve the exact same result as a RIGHT JOIN by simply swapping the tables and using a LEFT JOIN.
- Thinking it Through:
  - A RIGHT JOIN from Suppliers to Products says: "Give me every Product, and the matching Supplier if one exists."
  - A LEFT JOIN from Products to Suppliers says: "Give me every Product, and the matching Supplier if one exists."
  - They are logically identical!
- The Working Query for SQLite:
  - SELECT P.ProductName, S.SupplierName
  - FROM Products P -- We start with Products (the "right" table from before)
  - LEFT JOIN Suppliers S ON P.SupplierID = S.SupplierID;

• BASIC SQL JOINS: "FULL OUTER JOIN"

- What it does: A FULL OUTER JOIN combines a LEFT and RIGHT join. It returns all rows from both tables, matching them where possible. If there's no match for a row, the columns from the other table will be NULL.
- Analogy: The Annual Data Audit: The head office wants a complete data integrity report. This join gives you a master list showing:
  - Suppliers and the products they supply (the clean data).
  - Suppliers who supply no products (inactive suppliers).
  - Products that have no assigned supplier (a data entry error you need to fix!).
- The Query:
  - SELECT S.SupplierName, P.ProductName FROM Suppliers S FULL OUTER JOIN Products P ON S.SupplierID = P.SupplierID;
- This query reads as:
  - "Show me every supplier and every product. Where a supplier and product are linked by SupplierID, show them on the same line. If a supplier has no products, or a product has no supplier, still show it, but leave the other side blank (NULL)."

• SIMULATING A FULL OUTER JOIN IN SQLITE

- The Logic: A FULL OUTER JOIN is essentially the results of a LEFT JOIN and a RIGHT JOIN combined, with the duplicate matching rows removed. Since we can't use RIGHT JOIN, we simulate it this way:
  - Get all rows from a LEFT JOIN (Suppliers -> Products). This gives us every supplier and their matched products.
  - Use UNION.
  - Get the rows that would have appeared in a RIGHT JOIN but were missed by the LEFT JOIN. These are the products that have no supplier.

• THE WORKING QUERY FOR SQLITE

- -- Part 1: Get all Suppliers and their matched Products.
- SELECT S.SupplierName, P.ProductName
- FROM Suppliers S
- LEFT JOIN Products P ON S.SupplierID = P.SupplierID
- UNION
- -- Part 2: Get all Products that have no matching Supplier.
- SELECT S.SupplierName, P.ProductName
- FROM Products P
- LEFT JOIN Suppliers S ON P.SupplierID = S.SupplierID
- WHERE S.SupplierName IS NULL;

• THE "WHY" - THE NEED FOR AGGREGATION

- We can now combine detailed data. But what if we want to see the big picture and generate reports? We need to ask summary questions.
  - "How many customers do we have in each country?"
  - "Which supplier provides us with the most products?"
- Analogy: From a Sales Ledger to a Business Report: Your raw data is like a long ledger of every single sale. That's too much detail. A manager wants a report: "Show me total sales per country." To do this, you must group all sales by country and then sum their totals. This is aggregation.

• BASIC SQL QUERIES: "GROUP BY"

- What it does: The GROUP BY clause groups rows that have the same values into summary rows. It is used with aggregate functions like COUNT(), MAX(), SUM(), AVG().
- Task: Count the number of customers in each country.
  - SELECT Country, COUNT(CustomerID) AS NumberOfCustomers FROM Customers GROUP BY Country;
- This query reads as:
  - "From the Customers table, group all the rows by Country. For each of these groups, COUNT the number of customers and show me the Country and the final count."

• BASIC SQL QUERIES: "HAVING"

- What it does: The HAVING clause filters the results of a GROUP BY based on the aggregate function's result. WHERE filters rows before grouping; HAVING filters groups after grouping.
- Task: Find the countries with more than 5 customers.
  - SELECT Country, COUNT(CustomerID) AS NumberOfCustomers FROM Customers GROUP BY Country HAVING COUNT(CustomerID) > 5;
- This query reads as:
  - "First, group customers by country and count them, just like before. Then, from those resulting summary groups, only show me the ones HAVING a customer count greater than 5."

• HANDS-ON LAB: ADVANCED QUERIES

- Goal:
  - Use the online SQL playground (SQLite Online) to run advanced queries against the sample store database.
- Tool:
  - We will continue using SQLite Online with its pre-loaded database containing Customers, Products, Suppliers, etc.

STEP 1: RUN A "JOIN" AND "WHERE" QUERY

- Task: Find all products supplied by the supplier named 'Exotic Liquids'.
- In the SQL Editor, type the following query:
  - SELECT P.ProductName, P.Price FROM Products P INNER JOIN Suppliers S ON P.SupplierID = S.SupplierID WHERE S.SupplierName = 'Exotic Liquids’;
- This query tells the database:
  - "First, create a temporary combined table from Products and Suppliers. Then, from that combined data, only show me the product name and price for rows where the supplier's name is 'Exotic Liquids'."
- Click the "Run" button.

STEP 2: RUN A "JOIN" WITH "GROUP BY"

- Task: Count how many products each supplier provides.
- Delete the old query and type this new one in the SQL Editor:
  - SELECT S.SupplierName, COUNT(P.ProductID) AS NumberOfProducts FROM Suppliers S INNER JOIN Products P ON S.SupplierID = P.SupplierID GROUP BY S.SupplierName;
- Explanation:
  - This query joins the tables, then groups the results by supplier to COUNT the number of products for each one.
- Click "Run".

• STEP 3: RUN A "JOIN" WITH "GROUP BY", "HAVING", AND "ORDER BY"

- Task: Find suppliers who provide more than 2 products, and list them from most products to least.
- Modify your last query to add HAVING and ORDER BY clauses:
  - SELECT S.SupplierName, COUNT(P.ProductID) AS NumberOfProducts FROM Suppliers S INNER JOIN Products P ON S.SupplierID = P.SupplierID GROUP BY S.SupplierName HAVING COUNT(P.ProductID) > 2 ORDER BY NumberOfProducts DESC;
- Explanation:
  - This adds two final steps: the HAVING clause filters out suppliers with 2 or fewer products, and the ORDER BY clause sorts the final list.
- Click "Run".

• BONUS CHALLENGE

- Challenge 1: Finding Inactive Suppliers
  - Write a query to find the names of any suppliers who do not supply any products.
  - Hint: You will need a LEFT JOIN from Suppliers to Products and a WHERE clause that checks if the ProductID is NULL.
- Challenge 2: Multi-Join and Group
  - List all customers from 'London' and the total number of orders each has placed.
  - Hint: You will need to INNER JOIN the Customers and Orders tables, use a WHERE clause to filter by city, and then GROUP BY the customer's name.