- EXERCISES (22.08.2024)

-- Query 1

Select Customers, SUM(Price) AS TotalAmountSpent from Products

group by Customers;

-- Query 2

Select Customers, SUM(Price) AS TotalAmount from Products

group by Customers

having SUM(Price) > 1000;

-- Query 3

Select Category, COUNT(ProductId) As NumofProducts from Products

Group By Category

Having count(ProductId) > 5;

-- Query 4

Select category, COUNT(ProductId) AS TotalProducts from Products

Group by category;

-- Query 5

Select p.Customers, p.ProductId, SUM(o.Quantity\*p.Price) AS TotalSales

From Products p

JOIN Orders o ON p.ProductId = o.ProductId

Group By p.Customers, p.ProductId;

-- Stored Procedure with Insert Operation

CREATE PROCEDURE InsertCustomer

@CustomerID INT,

@CustomerName VARCHAR(100),

@BirthDate DATE

AS

BEGIN

INSERT INTO Customers (CustomerID, CustomerName, BirthDate)

VALUES (@CustomerID, @CustomerName, @BirthDate);

END;

-- Stored Procedure with Update Operation

CREATE PROCEDURE UpdateCustomer

@CustomerID INT,

@CustomerName VARCHAR(100),

@BirthDate DATE

AS

BEGIN

UPDATE Customers

SET CustomerName = @CustomerName,

BirthDate = @BirthDate

WHERE CustomerID = @CustomerID;

END;

-- Stored Procedure with Delete Operation

CREATE PROCEDURE DeleteCustomer

@CustomerID INT

AS

BEGIN

DELETE FROM Customers

WHERE CustomerID = @CustomerID;

END;

-- HANDS ON EXERCISES:

-- Exercise 1: Filtering Data using SQL Queries

SELECT \* FROM Products

WHERE category = 'Electronics' AND Price > 500;

-- Exercise 2: Total Aggregations using SQL Queries

SELECT SUM(Quantity) AS TotalQuantity from Orders;

-- Exercise 3: Group By Aggregations using SQL Queries

SELECT p.ProductName, p.ProductId, SUM(o.Quantity\*p.Price) AS TotalRevenue

FROM Orders o

JOIN Products p ON o.ProductId = p.ProductId

GROUP BY p.ProductId, p.ProductName;

-- Exercise 4: Order of Execution of SQL Queries

SELECT

Category,

COUNT(ProductId) AS ProductCount,

AVG(Price) AS AveragePrice

FROM

Products

WHERE

Price > 100

GROUP BY

Category

HAVING

COUNT(ProductId) > 0

ORDER BY

AveragePrice;

-- Exercise 5: Rules and Restrictions to Group and Filter Data in SQL Queries

SELECT

ProductId,

ProductName,

Category,

SUM(Price) AS TotalPrice

FROM

Products

GROUP BY

ProductId,

ProductName,

Category;

-- Exercise 6: Filter Data based on Aggregated Results using Group By and Having

SELECT

CustomerId,

COUNT(OrderID) AS OrderCount

FROM

Orders

GROUP BY

CustomerId

HAVING

COUNT(OrderID) > 5;

-- STORED PROCEDURES

-- Stored Procedure 1: Basic Stored Procedure

CREATE PROCEDURE GetAllCustomers

AS

BEGIN

SELECT \* FROM Customers;

END;

-- Stored Procedure 2: Stored Procedure with Input Parameter

CREATE PROCEDURE GetOrderDetailsByOrderID

@OrderID INT

AS

BEGIN

SELECT \* FROM Orders

WHERE OrderID = @OrderID;

END;

-- Stored Procedure 3: Stored Procedure with Multiple Input Parameters

CREATE PROCEDURE GetProductsByCategoryAndPrice

@Category VARCHAR(50),

@MinPrice DECIMAL(10,2)

AS

BEGIN

SELECT \* FROM Products

WHERE Category = @Category

AND Price >= @MinPrice;

END;

-- Stored Procedure 4: Stored Procedure with Insert Operation

CREATE PROCEDURE InsertNewProduct

@ProductName VARCHAR(100),

@Category VARCHAR(100),

@Price DECIMAL(10, 2),

@StockQuantity INT

AS

BEGIN

INSERT INTO Products (ProductName, Category, Price, StockQuantity)

VALUES (@ProductName, @Category, @Price, @StockQuantity);

END;

-- Stored Procedure 5: Stored Procedure with Update Operation

CREATE PROCEDURE UpdateCustomerEmail

@CustomerID INT,

@NewEmail VARCHAR(25)

AS

BEGIN

UPDATE Customers

SET Email = @NewEmail

WHERE CustomerID = @CustomerID;

END;

-- Stored Procedure 6: Stored Procedure with Delete Operation

CREATE PROCEDURE DeleteOrderByID

@OrderID INT

AS

BEGIN

DELETE FROM Orders

WHERE OrderID = @OrderID;

END;

-- Stored Procedure 7: Stored Procedure with Output Parameter

CREATE PROCEDURE GetTotalProductsInCategory

@Category VARCHAR(100),

@TotalProducts INT OUTPUT

AS

BEGIN

SELECT @TotalProducts = COUNT(\*)

FROM Products

WHERE Category = @Category;

END;

24th

**EXERCISE - 2**

1. Join the `Orders` and `Customers` tables to find the total order amount per customer and filter out customers who have spent less than $1,000.

**Query:** SELECT c.CustomerID, c.FirstName, c.LastName, SUM(o.TotalAmount) AS

             TotalSpent

 FROM Orders o

 JOIN Customers c ON o.CustomerID = c.CustomerID

 GROUP BY c.CustomerID, c.FirstName, c.LastName

 HAVING SUM(o.TotalAmount) >= 1000;

2. Create a cumulative sum of the `OrderAmount` for each customer to track the running total of how much each customer has spent.

**Query:** SELECT o.CustomerID, c.FirstName, c.LastName, o.OrderDate, o.TotalAmount,

  SUM(o.TotalAmount) OVER (PARTITION BY o.CustomerID ORDER BY

  o.OrderDate) AS RunningTotal

  FROM Orders o

  JOIN Customers c ON o.CustomerID = c.CustomerID;

3. Rank the customers based on the total amount they have spent, partitioned by city.

**Query:** SELECT c.CustomerID, c.City, SUM(o.TotalAmount) AS TotalSpent,

RANK() OVER (PARTITION BY c.City ORDER BY SUM(o.TotalAmount) DESC) AS CustomerRank

FROM Orders o

JOIN Customers c ON o.CustomerID = c.CustomerID

GROUP BY c.CustomerID, c.City;

4. Calculate the total amount of all orders (overall total) and the percentage each customer's total spending contributes to the overall total.

**Query:** WITH CustomerTotals AS (

SELECT c.CustomerID, SUM(o.TotalAmount) AS   TotalSpent

  FROM Orders o

    JOIN Customers c ON o.CustomerID = c.CustomerID

    GROUP BY c.CustomerID,

)

 SELECT CustomerID, TotalSpent,

       TotalSpent \* 100.0 / SUM(TotalSpent) OVER () AS PercentageOfTotal

 FROM CustomerTotals;

5. Rank all customers based on the total amount they have spent, without partitioning.

**Query:** SELECT c.CustomerID, c.FirstName, SUM(o.TotalAmount) AS TotalSpent,

         RANK() OVER (ORDER BY SUM(o.TotalAmount) DESC) AS CustomerRank

  FROM Orders o

  JOIN Customers c ON o.CustomerID = c.CustomerID

  GROUP BY c.CustomerID, c.FirstName;

6. Write a query that joins the `Orders` and `Customers` tables, calculates the average order amount for each city, and orders the results by the average amount in descending order.

**Query:** SELECT c.City, AVG(o.TotalAmount) AS AvgOrderAmount

  FROM Orders o

  JOIN Customers c ON o.CustomerID = c.CustomerID

  GROUP BY c.City

  ORDER BY AvgOrderAmount DESC;

7. Write a query to find the top 3 customers who have spent the most, using `ORDER BY` and `LIMIT`.

**Query:** SELECT c.CustomerID, c.FirstName, SUM(o.TotalAmount) AS TotalSpent

  FROM Orders o

  JOIN Customers c ON o.CustomerID = c.CustomerID

  GROUP BY c.CustomerID, c.FirstName,

  ORDER BY TotalSpent DESC

  LIMIT 3;

8. Write a query that groups orders by year (using `OrderDate`), calculates the total amount of orders for each year, and orders the results by year.

**Query:** SELECT YEAR(o.OrderDate) AS OrderYear, SUM(o.TotalAmount) AS TotalAmount

 FROM Orders o

 GROUP BY YEAR(o.OrderDate)

 ORDER BY OrderYear;

9. Write a query that ranks customers by their total spending, but only for customers located in "Mumbai". The rank should reset for each customer in "Mumbai".

**Query:** SELECT c.CustomerID, c.FirstName, SUM(o.TotalAmount) AS TotalSpent,

 RANK() OVER (ORDER BY SUM(o.TotalAmount) DESC) AS CustomerRank

 FROM Orders o

 JOIN Customers c ON o.CustomerID = c.CustomerID

 WHERE c.City = 'Mumbai'

 GROUP BY c.CustomerID, c.FirstName;

10. Write a query that calculates each customer's total order amount and compares it to the average order amount for all customers.

Query:   SELECT c.CustomerID, c.FirstName, c.LastName,

              SUM(o.TotalAmount) AS TotalSpent,

              SUM(o.TotalAmount) - AVG(SUM(o.TotalAmount)) OVER () AS DifferenceFromAvg

              FROM Orders o

              JOIN Customers c ON o.CustomerID = c.CustomerID

              GROUP BY c.CustomerID, c.FirstName, c.LastName;

28th



import pandas as pd

# Creating a new dataset

data = {

"Employee\_ID": [101, 102, 103, 104, 105, 106],

"Name": ["Rajesh", "Meena", "Suresh", "Anita", "Vijay", "Neeta"],

"Department": ["HR", "IT", "Finance", "IT", "Finance", "HR"],

"Age": [29, 35, 45, 32, 50, 28],

"Salary": [70000, 85000, 95000, 64000, 120000, 72000],

"City": ["Delhi", "Mumbai", "Bangalore", "Chennai", "Delhi", "Mumbai"]

}

df = pd.DataFrame(data)

print(df)

Exercise 1: Rename Columns

Rename the "Salary" column to "Annual Salary" and "City" to "Location".

Print the updated DataFrame.

Exercise 2: Drop Columns

Drop the "Location" column from the DataFrame.

Print the DataFrame after dropping the column.

Exercise 3: Drop Rows

Drop the row where "Name" is "Suresh".

Print the updated DataFrame.

Exercise 4: Handle Missing Data

Assign None to the "Salary" of "Meena".

Fill the missing "Salary" value with the mean salary of the existing employees.

Print the cleaned DataFrame.

Exercise 5: Create Conditional Columns

Create a new column "Seniority" that assigns "Senior" to employees aged 40 or above and "Junior" to employees younger than 40.

Print the updated DataFrame.

Exercise 6: Grouping and Aggregation

Group the DataFrame by "Department" and calculate the average salary in each department.

Print the grouped DataFrame.

import pandas as pd  
  
df=pd.read\_csv('employee.csv')  
print(df.head(3))  
print(df.info())  
print(df.describe())

Employee\_ID,Name,Department,Age,Salary,City  
101,Rajesh,HR,29,70000,Delhi  
102,Meena,IT,35,85000,Mumbai  
103,Suresh,Finance,45,95000,Bangalore  
104,Anita,IT,32,64000,Chennai  
105,Vijay,Finance,50,120000,Delhi  
106,Neeta,HR,28,72000,Mumbai

### \*\*Exercise 1: Creating DataFrame from Scratch\*\*

1. Create a DataFrame with the following columns: `"Product"`, `"Category"`, `"Price"`, and `"Quantity"`. Use the following data:

- Product: `['Laptop', 'Mouse', 'Monitor', 'Keyboard', 'Phone']`

- Category: `['Electronics', 'Accessories', 'Electronics', 'Accessories', 'Electronics']`

- Price: `[80000, 1500, 20000, 3000, 40000]`

- Quantity: `[10, 100, 50, 75, 30]`

2. Print the DataFrame.

### \*\*Exercise 2: Basic DataFrame Operations\*\*

1. Display the first 3 rows of the DataFrame.

2. Display the column names and index of the DataFrame.

3. Display a summary of statistics (mean, min, max, etc.) for the numeric columns in the DataFrame.

### \*\*Exercise 3: Selecting Data\*\*

1. Select and display the `"Product"` and `"Price"` columns.

2. Select rows where the `"Category"` is `"Electronics"` and print them.

### \*\*Exercise 4: Filtering Data\*\*

1. Filter the DataFrame to display only the products with a price greater than `10,000`.

2. Filter the DataFrame to show only products that belong to the `"Accessories"` category and have a quantity greater than `50`.

### \*\*Exercise 5: Adding and Removing Columns\*\*

1. Add a new column `"Total Value"` which is calculated by multiplying `"Price"` and `"Quantity"`.

2. Drop the `"Category"` column from the DataFrame and print the updated DataFrame.

### \*\*Exercise 6: Sorting Data\*\*

1. Sort the DataFrame by `"Price"` in descending order.

2. Sort the DataFrame by `"Quantity"` in ascending order, then by `"Price"` in descending order (multi-level sorting).

### \*\*Exercise 7: Grouping Data\*\*

1. Group the DataFrame by `"Category"` and calculate the total quantity for each category.

2. Group by `"Category"` and calculate the average price for each category.

### \*\*Exercise 8: Handling Missing Data\*\*

1. Introduce some missing values in the `"Price"` column by assigning `None` to two rows.

2. Fill the missing values with the mean price of the available products.

3. Drop any rows where the `"Quantity"` is less than `50`.

### \*\*Exercise 9: Apply Custom Functions\*\*

1. Apply a custom function to the `"Price"` column that increases all prices by 5%.

2. Create a new column `"Discounted Price"` that reduces the original price by 10%.

### \*\*Exercise 10: Merging DataFrames\*\*

1. Create another DataFrame with columns `"Product"` and `"Supplier"`, and merge it with the original DataFrame based on the `"Product"` column.

### \*\*Exercise 11: Pivot Tables\*\*

1. Create a pivot table that shows the total quantity of products for each category and product combination.

### \*\*Exercise 12: Concatenating DataFrames\*\*

1. Create two separate DataFrames for two different stores with the same columns (`"Product"`, `"Price"`, `"Quantity"`).

2. Concatenate these DataFrames to create a combined inventory list.

### \*\*Exercise 13: Working with Dates\*\*

1. Create a DataFrame with a `"Date"` column that contains the last 5 days starting from today.

2. Add a column `"Sales"` with random values for each day.

3. Find the total sales for all days combined.

### \*\*Exercise 14: Reshaping Data with Melt\*\*

1. Create a DataFrame with columns `"Product"`, `"Region"`, `"Q1\_Sales"`, `"Q2\_Sales"`.

2. Use `pd.melt()` to reshape the DataFrame so that it has columns `"Product"`, `"Region"`, `"Quarter"`, and `"Sales"`.

### \*\*Exercise 15: Reading and Writing Data\*\*

1. Read the data from a CSV file named `products.csv` into a DataFrame.

2. After performing some operations (e.g., adding a new column or modifying values), write the DataFrame back to a new CSV file named `updated\_products.csv`.

### \*\*Exercise 16: Renaming Columns\*\*

1. Given a DataFrame with columns `"Prod"`, `"Cat"`, `"Price"`, `"Qty"`, rename the columns to `"Product"`, `"Category"`, `"Price"`, and `"Quantity"`.

2. Print the renamed DataFrame.

### \*\*Exercise 17: Creating a MultiIndex DataFrame\*\*

1. Create a DataFrame using a MultiIndex (hierarchical index) with two levels: `"Store"` and `"Product"`. The DataFrame should have columns `"Price"` and `"Quantity"`, representing the price and quantity of products in different stores.

2. Print the MultiIndex DataFrame.

### \*\*Exercise 18: Resample Time-Series Data\*\*

1. Create a DataFrame with a `"Date"` column containing a range of dates for the past 30 days and a `"Sales"` column with random values.

2. Resample the data to show the total sales by week.

### \*\*Exercise 19: Handling Duplicates\*\*

1. Given a DataFrame with duplicate rows, identify and remove the duplicate rows.

2. Print the cleaned DataFrame.

### \*\*Exercise 20: Correlation Matrix\*\*

1. Create a DataFrame with numeric data representing different features (e.g., `"Height"`, `"Weight"`, `"Age"`, `"Income"`).

2. Compute the correlation matrix for the DataFrame.

3. Print the correlation matrix.

### \*\*Exercise 21: Cumulative Sum and Rolling Windows\*\*

1. Create a DataFrame with random sales data for each day over the last 30 days.

2. Calculate the cumulative sum of the sales and add it as a new column `"Cumulative Sales"`.

3. Calculate the rolling average of sales over the past 7 days and add it as a new column `"Rolling Avg"`.

### \*\*Exercise 22: String Operations\*\*

1. Create a DataFrame with a column `"Names"` containing values like `"John Doe"`, `"Jane Smith"`, `"Sam Brown"`.

2. Split the `"Names"` column into two separate columns: `"First Name"` and `"Last Name"`.

3. Convert the `"First Name"` column to uppercase.

### \*\*Exercise 23: Conditional Selections with `np.where`\*\*

1. Create a DataFrame with columns `"Employee"`, `"Age"`, and `"Department"`.

2. Create a new column `"Status"` that assigns `"Senior"` to employees aged 40 or above and `"Junior"` to employees below 40 using `np.where()`.

### \*\*Exercise 24: Slicing DataFrames\*\*

1. Given a DataFrame with data on `"Products"`, `"Category"`, `"Sales"`, and `"Profit"`, slice the DataFrame to display:

- The first 10 rows.

- All rows where the `"Category"` is `"Electronics"`.

- Only the `"Sales"` and `"Profit"` columns for products with sales greater than 50,000.

### \*\*Exercise 25: Concatenating DataFrames Vertically and Horizontally\*\*

1. Create two DataFrames with identical columns `"Employee"`, `"Age"`, `"Salary"`, but different rows (e.g., one for employees in `"Store A"` and one for employees in `"Store B"`).

2. Concatenate the DataFrames vertically to create a combined DataFrame.

3. Now create two DataFrames with different columns (e.g., `"Employee"`, `"Department"` and `"Employee"`, `"Salary"`) and concatenate them horizontally based on the common `"Employee"` column.

### \*\*Exercise 26: Exploding Lists in DataFrame Columns\*\*

1. Create a DataFrame with a column `"Product"` and a column `"Features"` where each feature is a list (e.g., `["Feature1", "Feature2"]`).

2. Use the `explode()` method to create a new row for each feature in the list, so each product-feature pair has its own row.

### \*\*Exercise 27: Using `.map()` and `.applymap()`\*\*

1. Given a DataFrame with columns `"Product"`, `"Price"`, and `"Quantity"`, use `.map()` to apply a custom function to increase `"Price"` by 10% for each row.

2. Use `.applymap()` to format the numeric values in the DataFrame to two decimal places.

### \*\*Exercise 28: Combining `groupby()` with `apply()`\*\*

1. Create a DataFrame with `"City"`, `"Product"`, `"Sales"`, and `"Profit"`.

2. Group by `"City"` and apply a custom function to calculate the profit margin (Profit/Sales) for each city.

### \*\*Exercise 29: Creating a DataFrame from Multiple Sources\*\*

1. Create three different DataFrames from different sources (e.g., CSV, JSON, and a Python dictionary).

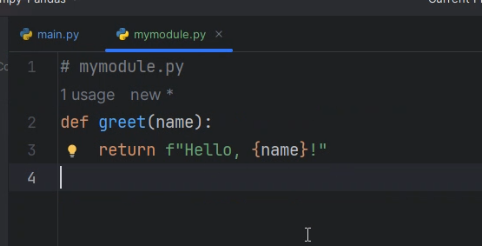
2. Merge the DataFrames based on a common column and create a consolidated report.

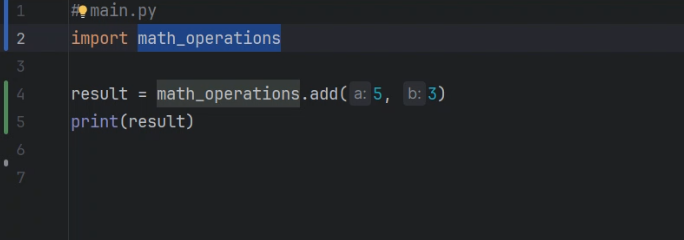
### \*\*Exercise 30: Dealing with Large Datasets\*\*

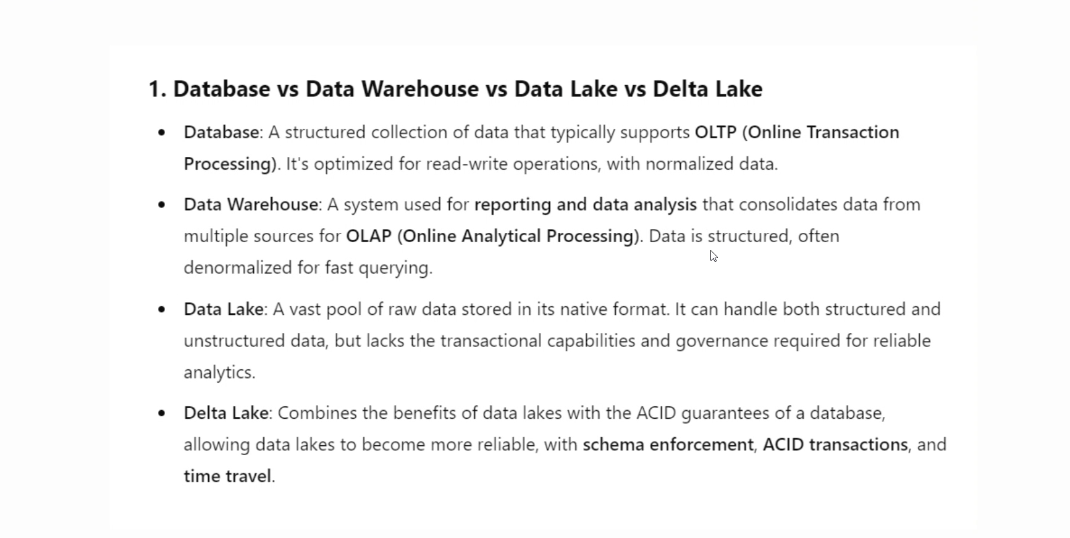
1. Create a large DataFrame with 1 million rows, representing data on `"Transaction ID"`, `"Customer"`, `"Product"`, `"Amount"`, and `"Date"`.

2. Split the DataFrame into smaller chunks (e.g., 100,000 rows each), perform a simple analysis on each chunk (e.g., total sales), and combine the results.

30th







---