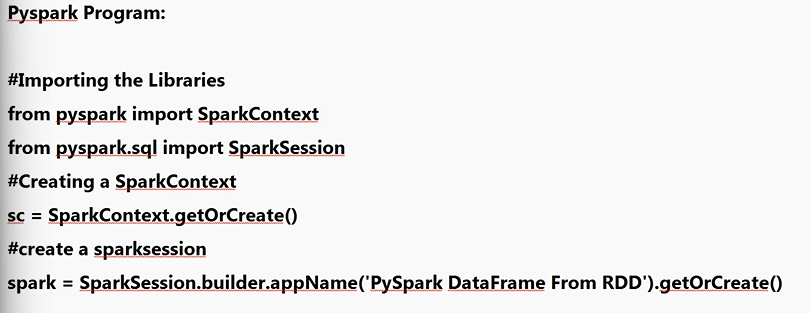
To write a code we need to start with these codes

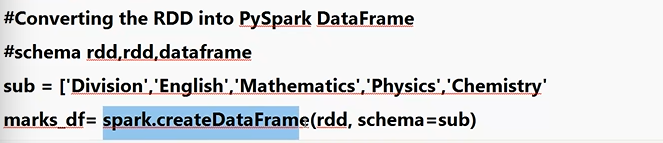


1st 4 lines wii be same for call code

Next:



Next:



## Spark basics

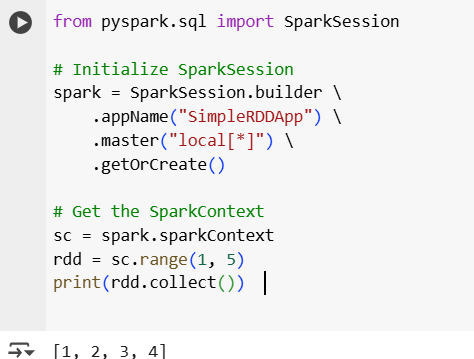
## **Spark context**

pyspark.SparkContext is an entry point to the PySpark functionality that is used to communicate with the cluster and to create an RDD, accumulator, and broadcast variables. The Spark driver program creates and uses SparkContext to connect to the cluster manager to submit PySpark jobs, and know what resource manager (YARN, Mesos, or Standalone) to communicate to. It is the heart of the PySpark application.



You can stop the SparkContext by calling the stop() method.If you want to create another, you need to shutdown it first by using stop() method and create a new SparkContext.

**Create PySpark RDD:**Once you have a SparkContext object, you can create a PySpark RDD in several ways. Below I have used the range() function.



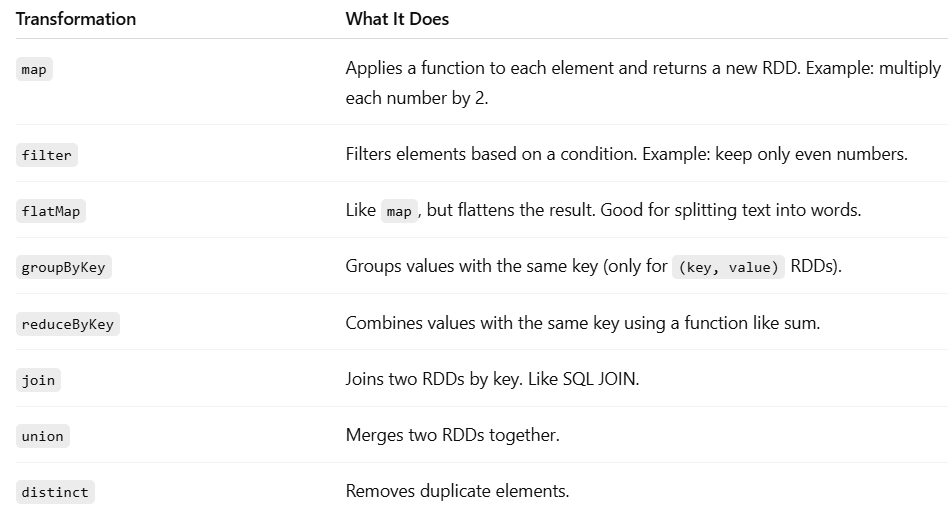
We can create rdd in three ways

- From a Text File

- From an existing Python collection

- By transforming another RDD

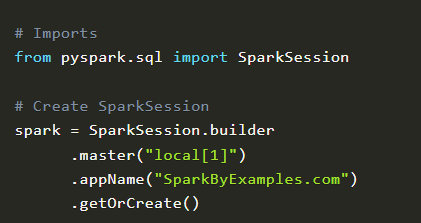
RDD Operations:



RDD (Resilient Distributed Dataset) is a core building block of PySpark. It is a fault-tolerant, immutable, distributed collection of objects. Immutable means that once you create an RDD, you cannot change it. The data within RDDs is segmented into logical partitions, allowing for distributed computation across multiple nodes within the cluster.

### **Step 1: Initialize SparkSession**

Before you can use RDDs, you need to create a SparkSession (which internally sets up SparkContext).



# Get SparkContext - add this in the last

sc = spark.sparkContext

### **📌 What does each method mean?**

* master("local[x]"): Runs Spark locally with x CPU cores.
* appName("..."): Sets your application name.
* getOrCreate(): Reuses an existing session or creates a new one.

### Using sparkContext.parallelize() By using parallelize() function of SparkContext ([sparkContext.parallelize()](https://sparkbyexamples.com/pyspark/pyspark-parallelize-create-rdd/) ) you can create an RDD. This function loads the existing collection from your driver program into parallelizing RDD. This method of creating an RDD is used when you already have data in memory that is either loaded from a file or from a database. and all data must be present in the driver program prior to creating RDD.

## **4. RDD Creation**

You can create RDDs in **two main ways**:

1. From a **Python collection** using .parallelize()
2. By **loading files** (like .txt, .csv, etc.) using .textFile() or .wholeTextFiles()

### **What is parallelize() in PySpark?**

parallelize() is a method used to **create an RDD from an existing Python collection (like a list or array)** that is already loaded in memory (in your driver program).

### **💡 When to use it?**

Use parallelize() when:

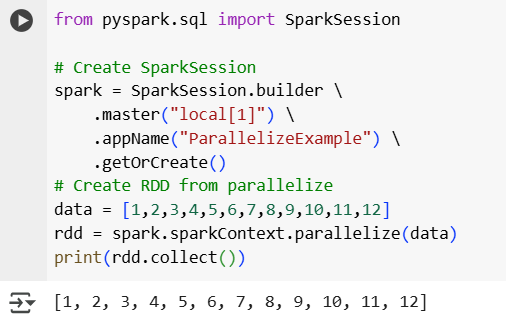
* You already have data in Python (like a list or result from a DB/file).
* You want to distribute that data across multiple nodes for parallel processing.

### **✅ Syntax**

rdd = spark.sparkContext.parallelize(your\_data, num\_partitions)

* your\_data: Your list or array.
* num\_partitions *(optional)*: How many chunks to split the data into.

### **🧪 Example:**



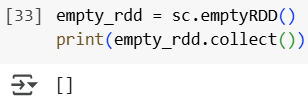
### **✅ Ways to Create an RDD:**

1. **From Python collection (in-memory) - same example use for parallelize**
2. **From external storage (disk)**

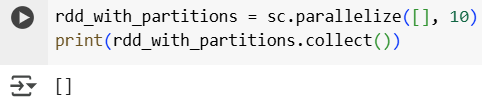
****

1. **Empty RDDs**

**Create an Empty RDD**

****

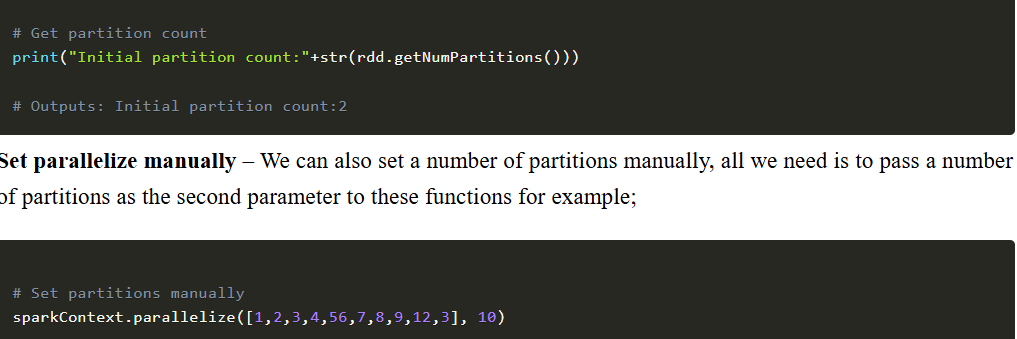
**Create an Empty RDD with 10 partitions**

****

## **RDD Partitions** When we use parallelize(), textFile() or wholeTextFiles() methods of SparkContext to initiate RDD, it automatically splits the data into partitions based on resource availability.

getNumPartitions() – This is an RDD function that returns a number of partitions your dataset split int

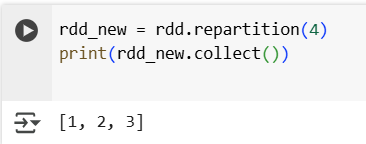
**Check number of partitions:**

****

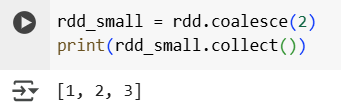
## **Repartition and Coalesce**

Sometimes, we may need to [repartition the RDD](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/), PySpark provides two ways to repartition; first using repartition() method, which shuffles data from all nodes also called full shuffle and second [coalesce()](https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-coalesce/) method which [shuffles](https://sparkbyexamples.com/spark/spark-shuffle-partitions/) data from minimum nodes, for examples if you have data in 4 partitions and doing coalesce(2) moves data from just 2 nodes.

### **Repartition (full shuffle, expensive):**



### **Coalesce (less shuffling, faster for reducing partitions):**



Note: repartition() or coalesce() methods also return a new RDD.

## **PySpark RDD Operations**

RDD operations are the core transformations and actions performed on RDDs

### **Two Types:**

**Transformations** – Create a new RDD (lazy, nothing runs until an action is called).

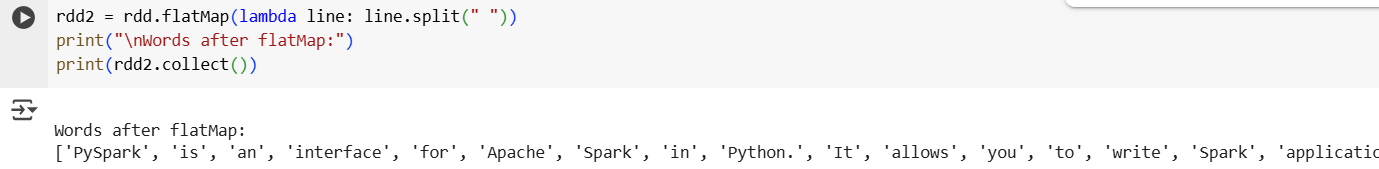
**Actions** – Trigger computation and return results (or save data).

Each line from the file becomes one RDD element.

### **🪵 Step 1: Load a text file**

* Each line from the file becomes one RDD element.

### **✂️ Step 2: flatMap() – Split lines into words**



* **Purpose**: Splits each line into individual words.
* **Why flatMap?**: It flattens the result. For example:  
  + Line: "Hello world" → ["Hello", "world"]
  + Normal map() would return a list of lists.

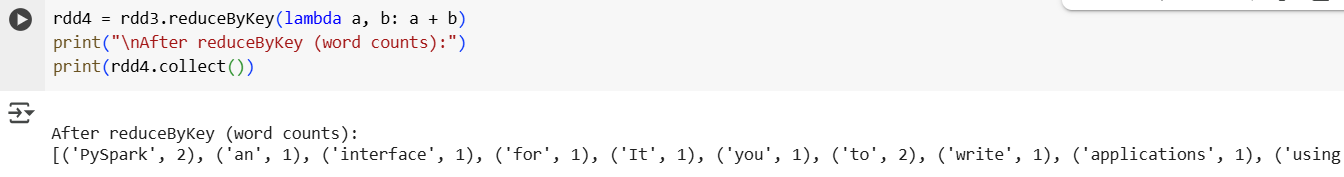
flatMap() returns a flat list: ["Hello", "world"]

### **➕ Step 3: map() – Add (word, 1) to each word**



* Converts each word into a key-value pair:  
   'word' → ('word', 1)

### **🧮 Step 4: reduceByKey() – Count words**



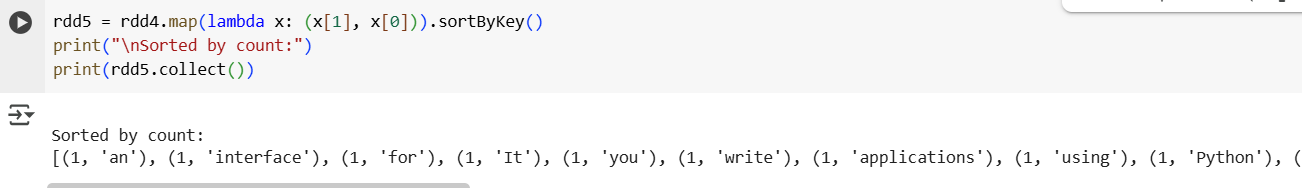
### **Purpose**: Adds up values for each word (key).

### **Example**:

### [("PySpark", 1), ("PySpark", 1)] → ("PySpark", 2)

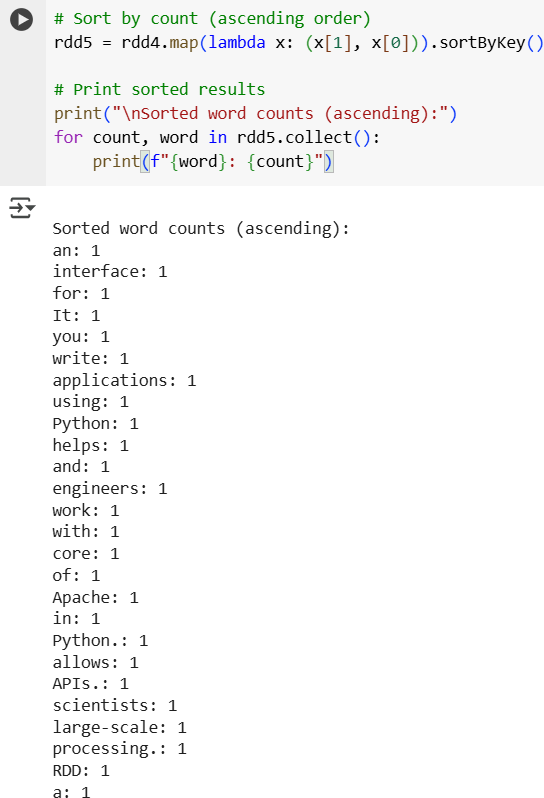
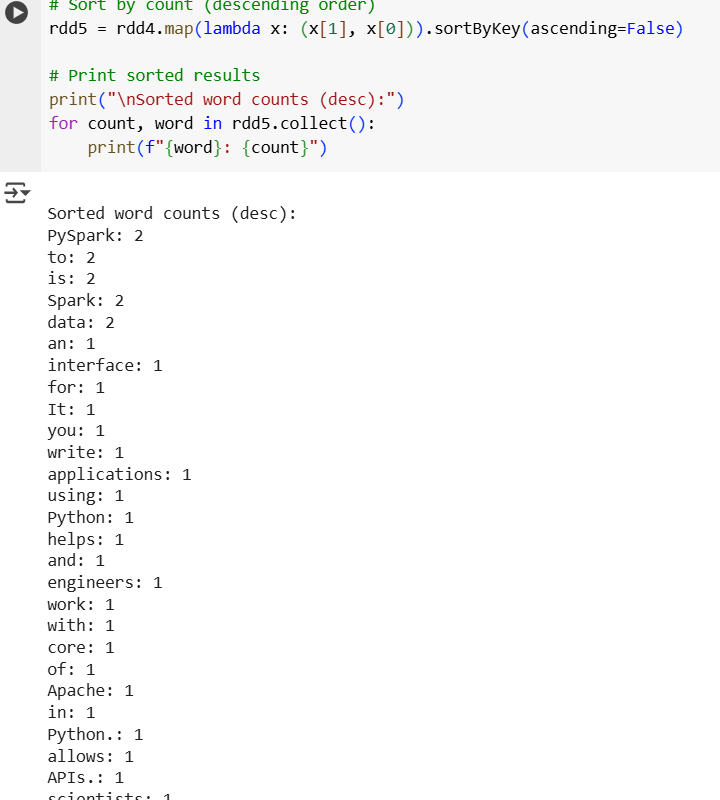
### **Efficient**: Combines values on the same partition before shuffling across the network.

### **Step 5: map() to flip key-value**



* **Purpose**: Swaps from (word, count) to (count, word)
* **Why?**: So we can sort by count (which is now the key).

### **🔢 Step 6: sortByKey() – Sort by frequency**



* First switch from (word, count) → (count, word)
* Then sort by count (key)
* Final output will be: lowest → highest frequency

### **🖨️ Display Results**

print(rdd5.collect())  
(action) - count(),first(),max(),reduce(),take(),collect(),saveAsTextFile()

## **PYSPARK DATAFRAME**

A DataFrame in PySpark is:

* A distributed collection of data organized into rows and columns.
* Similar to a table in a database or a Pandas DataFrame.
* Stored across multiple machines in a Spark cluster, which allows Spark to process big data in parallel.

💡 Difference from Pandas:

* Pandas DataFrame: Runs on one machine, data must fit in memory.
* PySpark DataFrame: Runs on many machines in parallel, handles huge datasets.

### **Why PySpark DataFrame is Faster than Pandas**

* PySpark: Uses cluster computing and parallel processing → good for large datasets.
* Pandas: Single-machine in-memory processing → better for small to medium datasets.

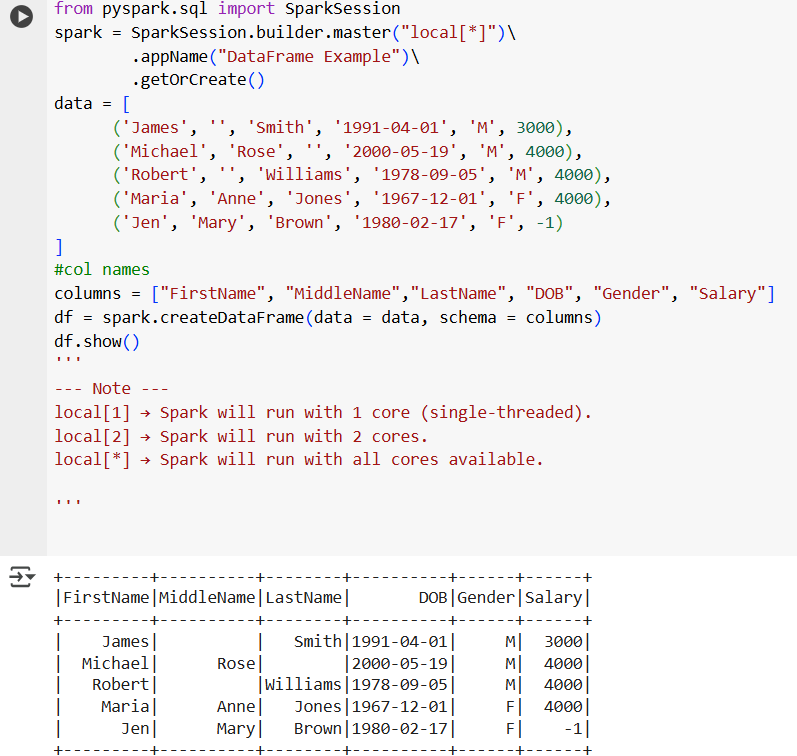
✅ If you have GBs or TBs of data, use PySpark.

✅ If you have MBs of data, Pandas is enough.

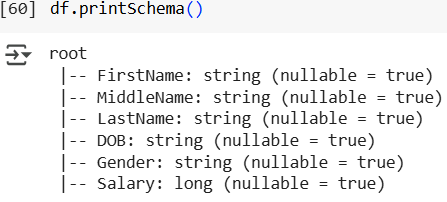
### **Creating a DataFrame** Ways to create Dataframe

* Create DataFrame from RDD  
   - Using to DF()  
   - Using createDataFrame()
* Create DataFrame from Files  
   - From CSV  
   - From Text File  
   - From JSON File
* Create DataFrame From Python List

**Create DataFrame from Python List**

****

**Check schema (column names & data types):**

****

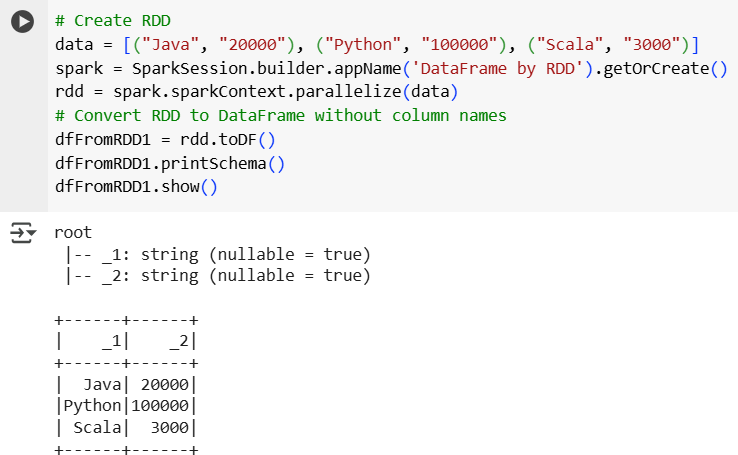
**Create DataFrame from RDD**

***Using toDF()***

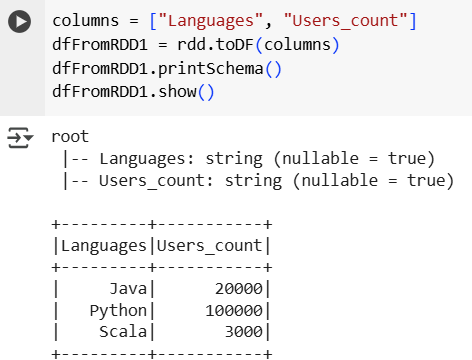
If you already have an RDD, you can convert it into a DataFrame using toDF().

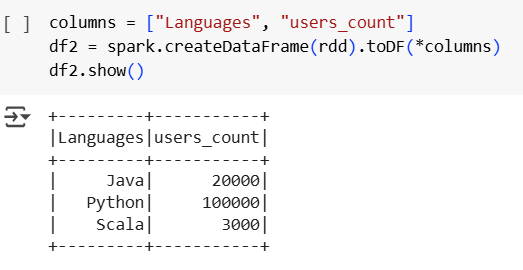
Note: In PySpark, parallelize() is a method of SparkContext that is used to create an RDD (Resilient Distributed Dataset) from a Python collection (like a list).

It takes your local data and distributes it across the Spark cluster (or CPU cores if running locally), allowing parallel processing.



**Adding Column Names**

****

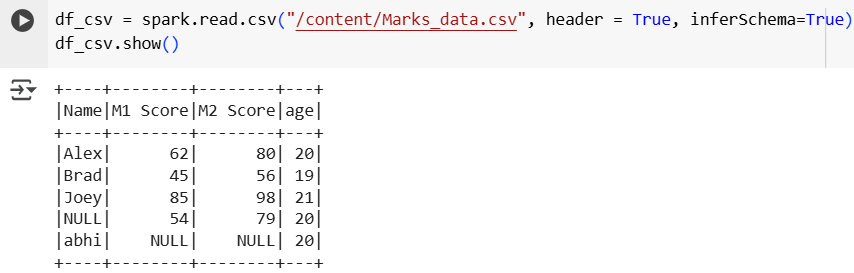
**Using createDataFrame()  
**

**Create DataFrame from Files**

**From CSV**

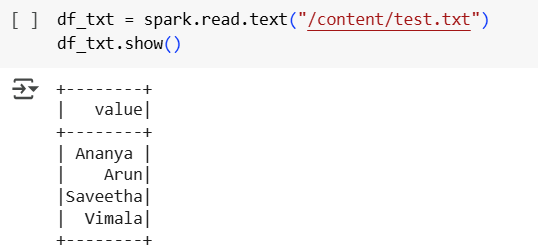
header=True → uses the first row as column names.

inferSchema=True → automatically detects column data types.



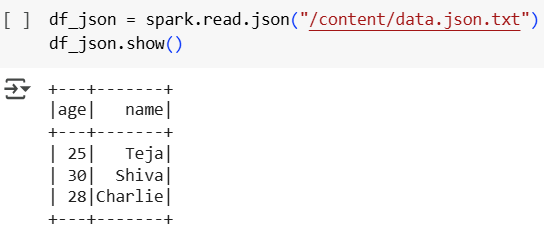
**From Text File**

This creates a single column DataFrame with column name value.



**From JSON File**

Spark automatically maps JSON keys to columns.



**Key Features of PySpark DataFrames**

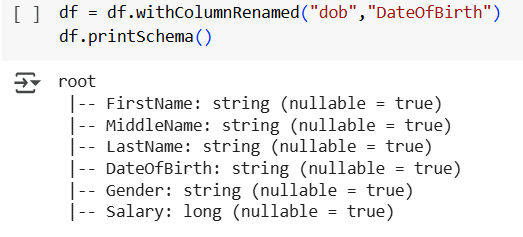
* Distributed → works on multiple machines.
* Optimized → uses Spark's engine for speed.
* Schema-aware → knows column names and data types.
* Supports SQL → you can run SQL queries using df.createOrReplaceTempView() and spark.sql().

### **DataFrame Operations**

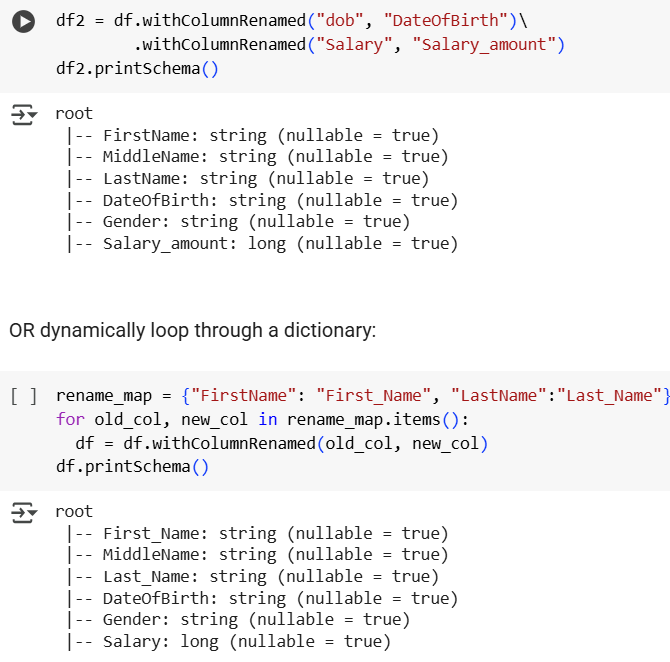
**1. RENAMING**

##### **Rename a Single Column**

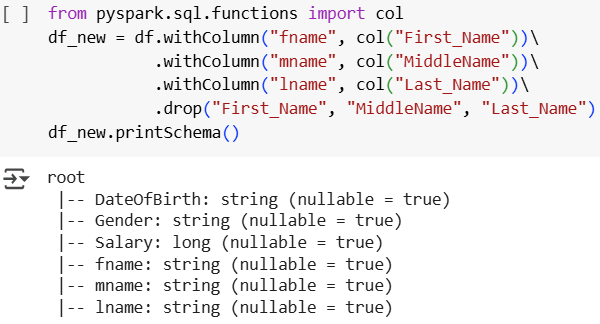
The simplest way is to use withColumnRenamed().

 (from dob to dateofbirth)

##### **Rename Multiple Columns**

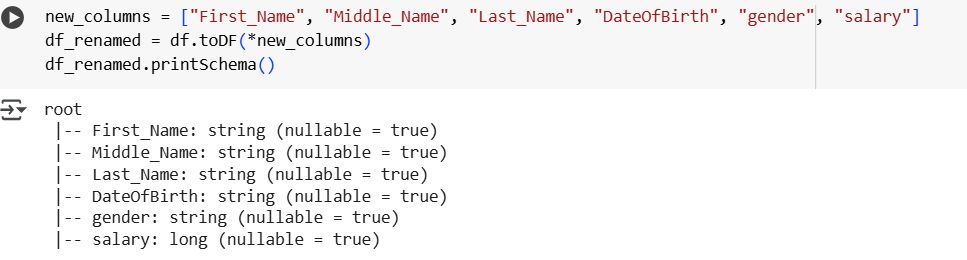
You can chain multiple withColumnRenamed() calls:

##### **Rename Nested Columns using withColumn()**



##### **Rename All Columns using toDF()**

If your DataFrame is flat (not nested), the easiest way is toDF():



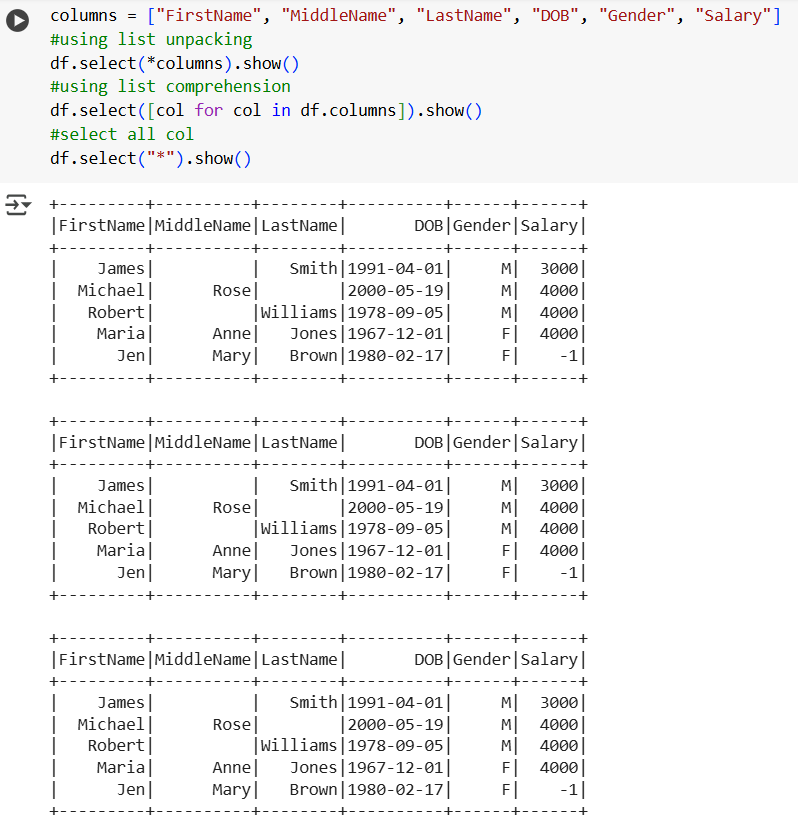
#### **Selecting**

* select() is a transformation function in PySpark DataFrame.
* It is used to select specific columns (single, multiple, by index, or nested).
* It always returns a new DataFrame (because DataFrames in Spark are immutable).

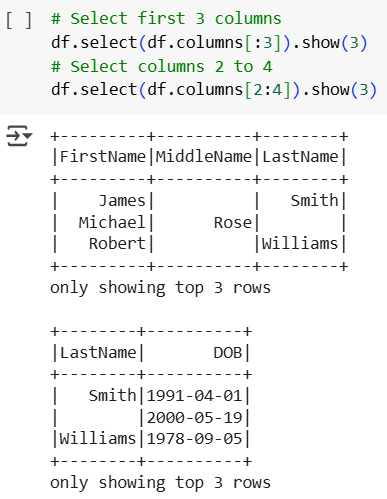
##### **Select Single or Multiple Columns**

##### **Select All Columns from a List**

If you have a list of column names, you can unpack it:



##### **Select Columns by Index**



#### **Filtering**

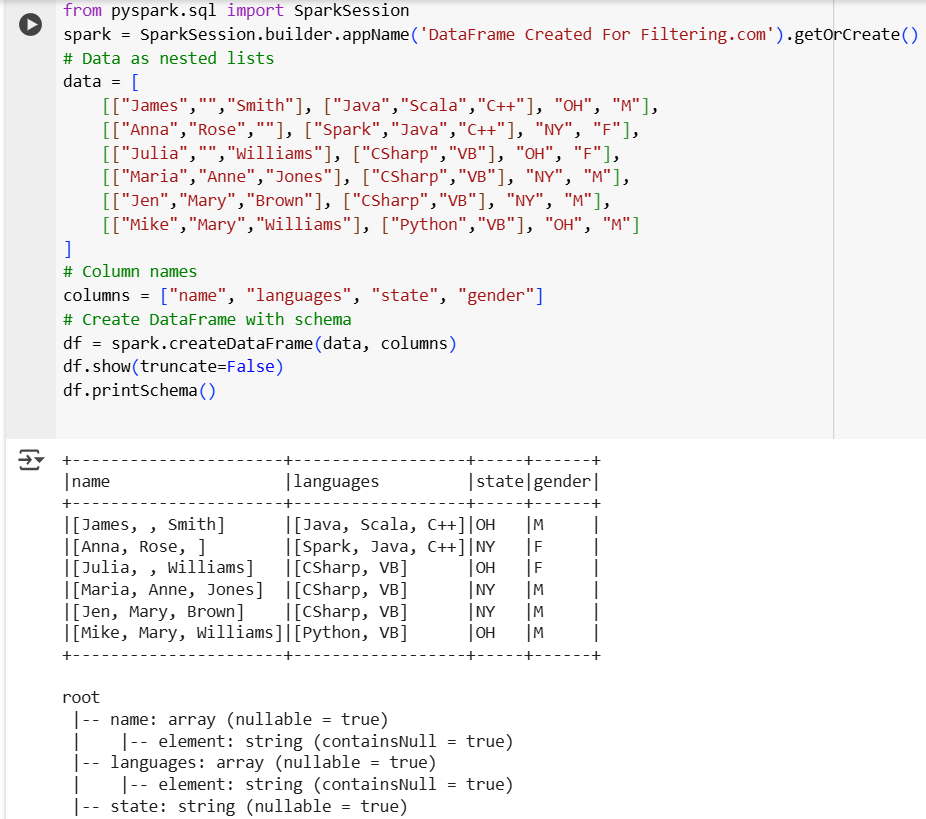
* filter() is used to filter rows in a DataFrame based on a condition.
* It is similar to the SQL WHERE clause.
* Returns a **new DataFrame**, doesn’t change the original.
* You can use **.filter()** or **.where()** — both do the same thing.

**Syntax:**df.filter(condition)

df.where(condition) #same as filter

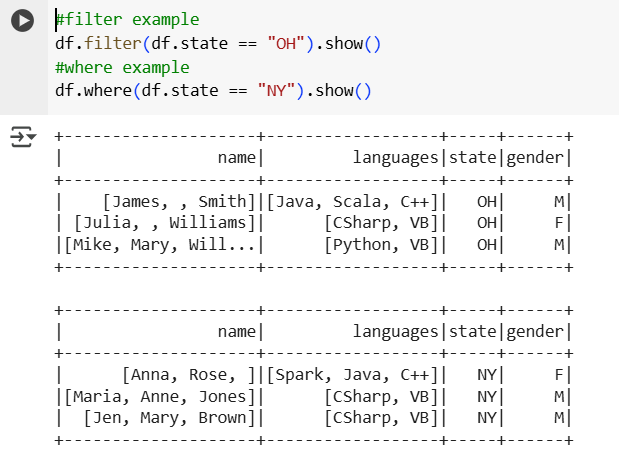
condition can be:

* A column expression
* A SQL expression (string)
* A function like startswith(), like(), isin(), etc.

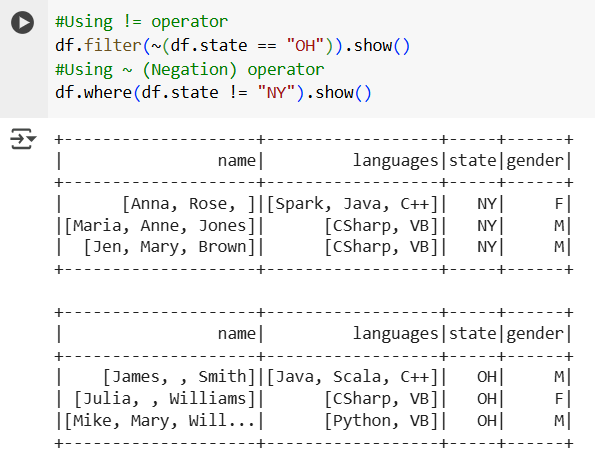
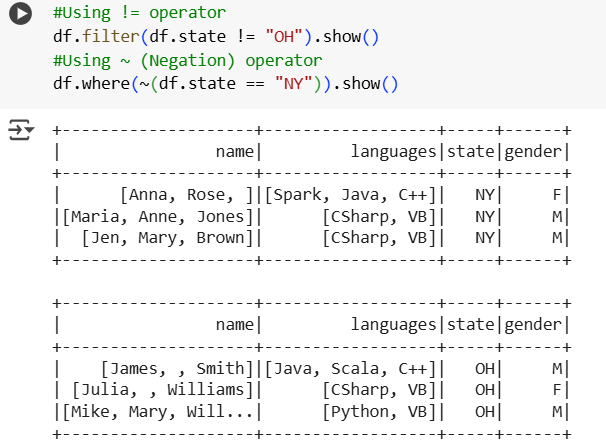


**1. Filter with Column Expressions**

##### **Filter using equal condition**

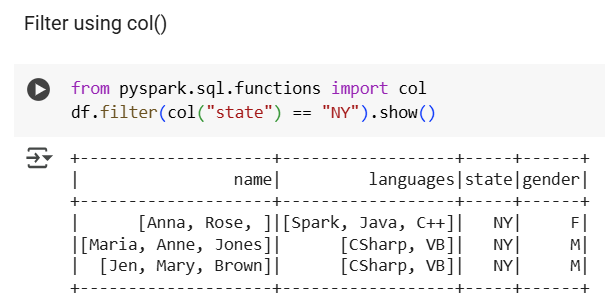
****

##### **Filter using not equal**

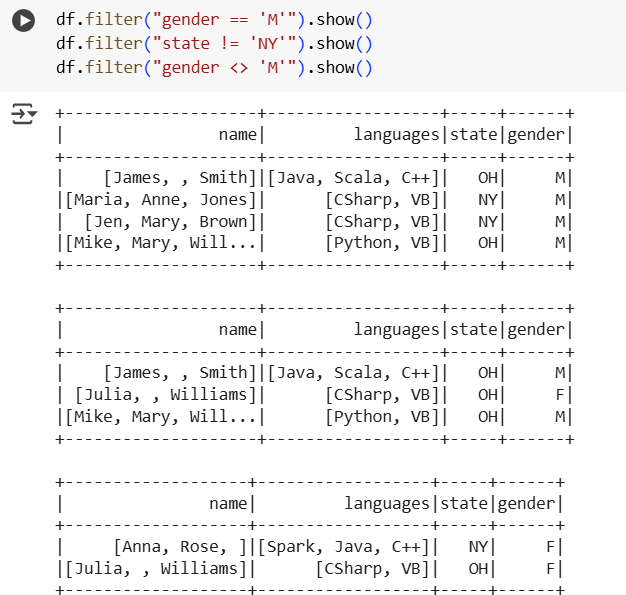


**Note : If you use this both ( ! ~ ) both operators can be used for filter and where**

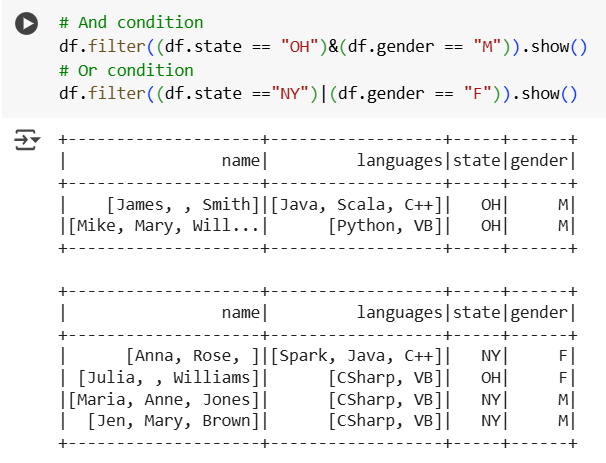
**2. Filter using SQL Expression**

****

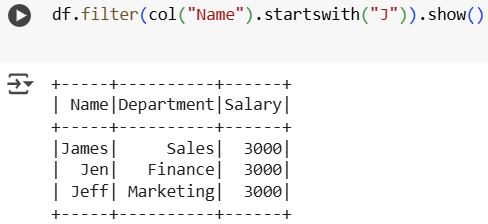
**Note:** In PySpark, <> is just another way of writing "not equal to", just like !=.

****

###### **3. Filter using multiple conditions - AND Condition, OR Condition**

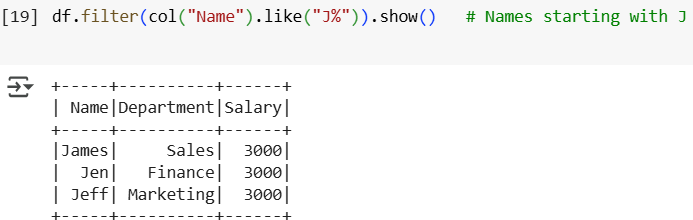


**4. Filter with functions - startswith()**, **like()** – SQL style pattern match,**isin()** – IN condition

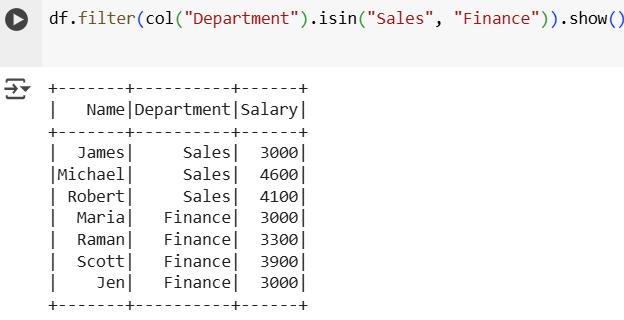


1. starts with() :

### 

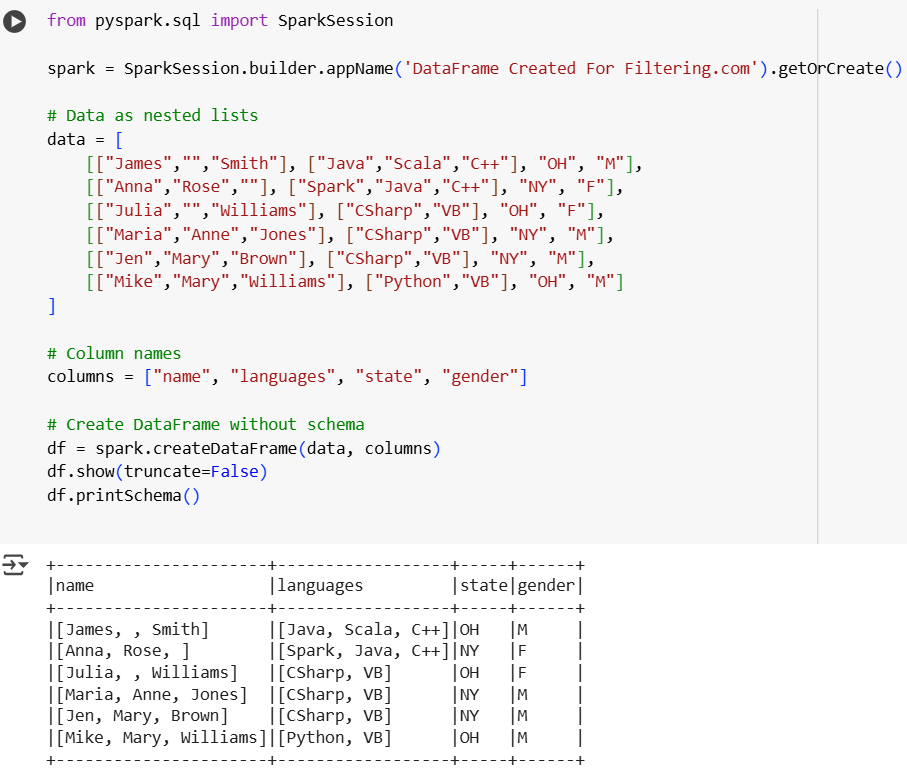


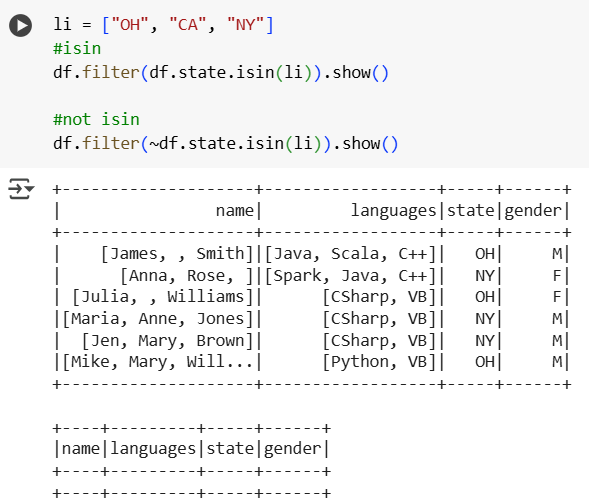
2.like() :



3.isin() :

##### **Filter using list values**

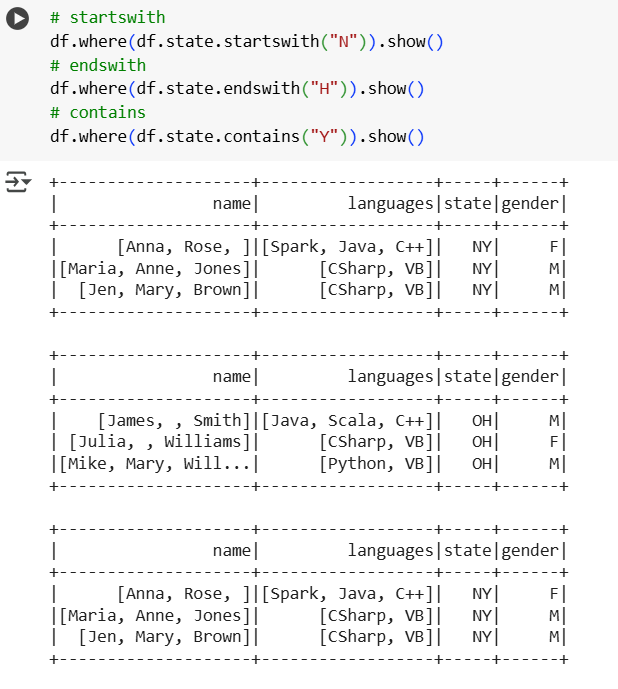




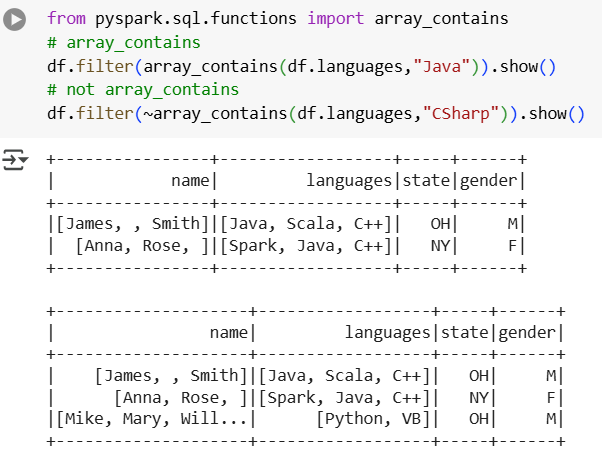
##### 

##### **Filter using string functions**

Using startwith, endwith, contain function

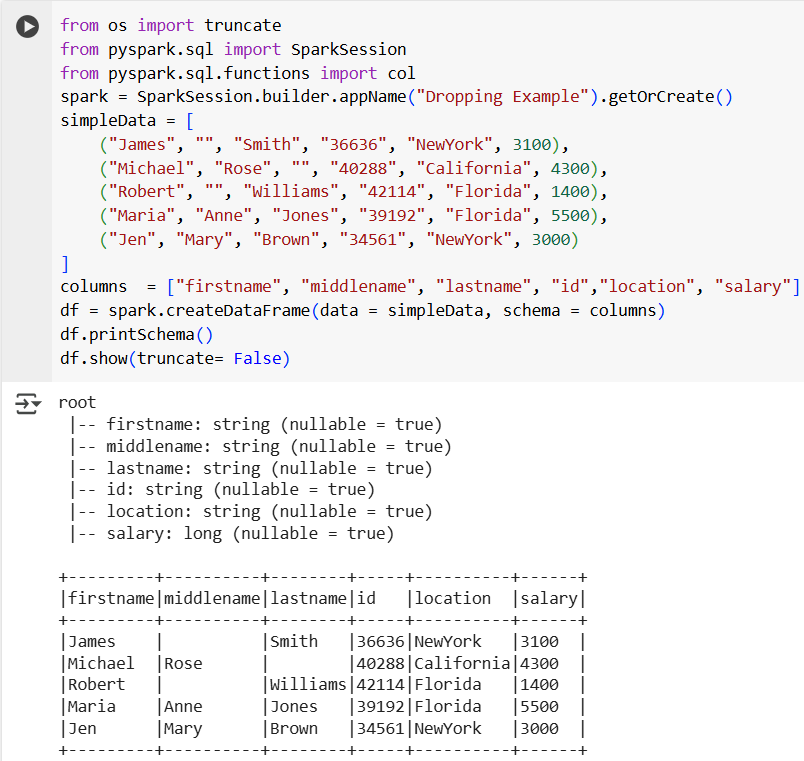


##### **Filter using array column**



# PySpark Distinct to Drop Duplicate Rows

#### **Dropping**



##### **Drop a Single Column**

In PySpark, if you want to remove (drop) one column from a DataFrame, you can use the .drop() method.  
Syntax: **df.drop(column\_name)**

But how you refer to the column can be done **in 3 different ways**:

## **1. Drop by String Name**

df.drop("firstname")

* "firstname" is passed as a **string**.
* This is the most common way.
* Works well when column names are simple and known.

✅ **Use when**: You just want to drop a column by name, plain and simple.

## **2. Drop by Column Object**

from pyspark.sql.functions import col

df.drop(col("firstname"))

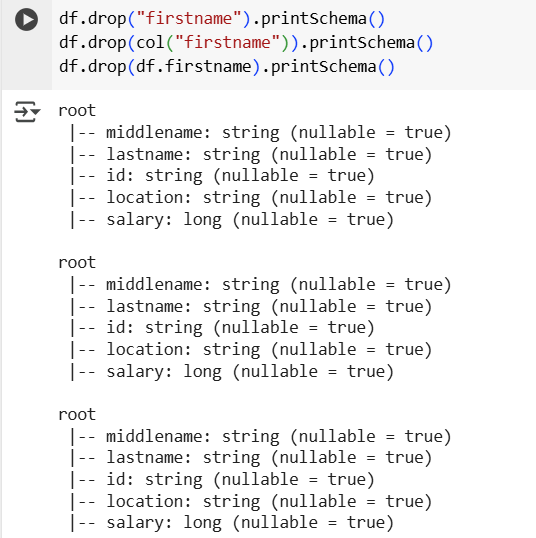
* col("firstname") returns a **column object**.
* You are explicitly telling PySpark: “Here’s a column object, drop it.”
* Useful when combining with conditions or other transformations.

✅ **Use when**: You already use col() for other expressions and want consistency.

## **3. Drop by DataFrame.column Reference**

df.drop(df.firstname)

* df.firstname directly accesses the column object from the DataFrame.
* This is more "Pythonic" but less flexible than using col().
* It’s like saying: “Hey, drop this column right from my DataFrame.”

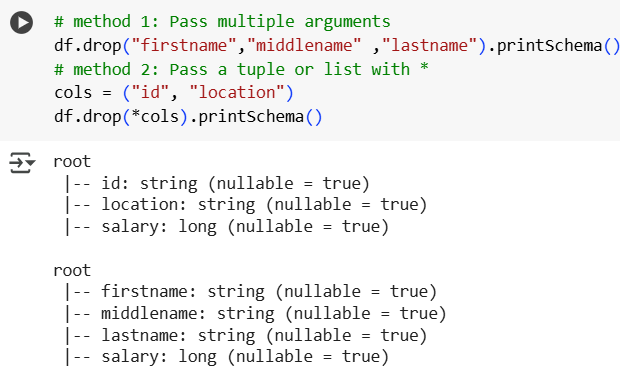
✅ **Use when**: You prefer dot notation or are already working with df.column\_name.

**EXAMPLE :**

##### **Drop Multiple Columns**

We can drop column in two ways

* Pass multiple arguments - Drops both firstname and lastname columns.
* Pass a tuple or list with \* - The \* unpacks the list and passes each element as a separate argument to drop().This is cleaner when you already have a list of column names.



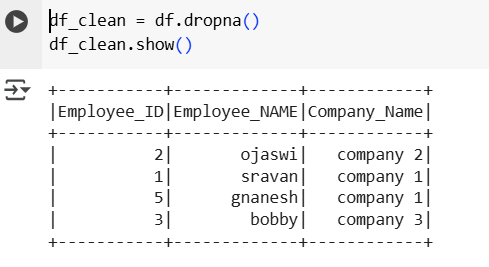
##### **Drop Rows**



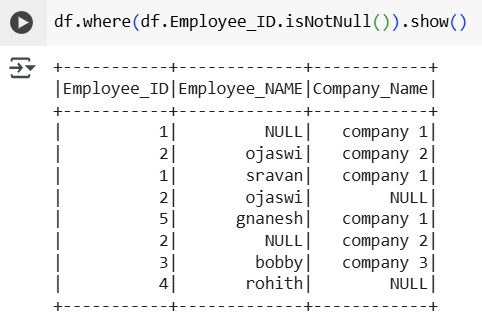
**Drop Rows with Null or Missing Values**

**1) Using dropna() :**

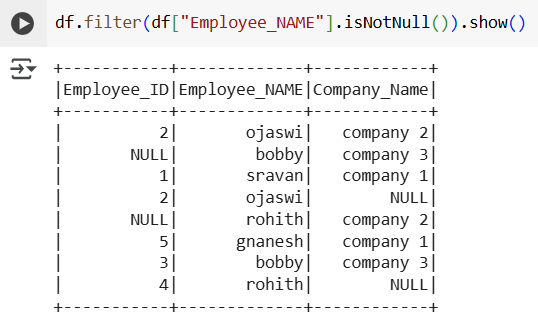
The dropna() method is used to remove rows that contain any null (missing) values in the DataFrame. It’s handy for quick cleanup when you want to keep only fully complete records



**2) Using isNotNull() :**  
The isNotNull() method is used to **filter out null values** in a DataFrame.It's commonly used with the .filter() or .where() clause to **keep only rows where a specific column is *not null***.



this is using where()

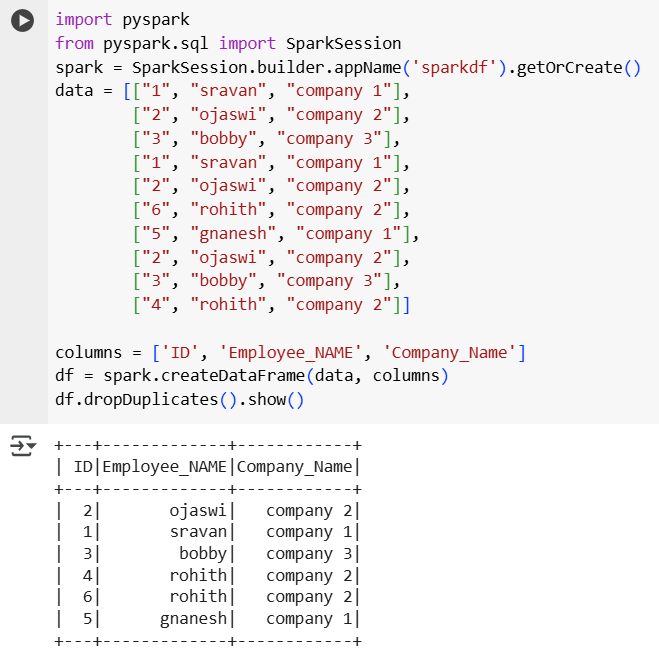


this is for filter ()

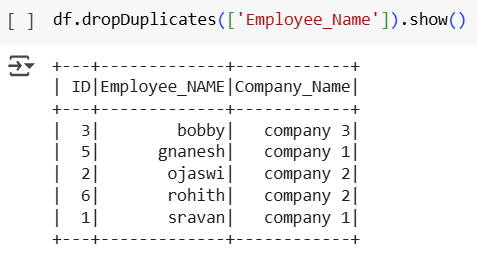
###### **Drop duplicate rows**

Duplicate rows mean rows are the same among the dataframe, we are going to remove those rows by using dropDuplicates() function.

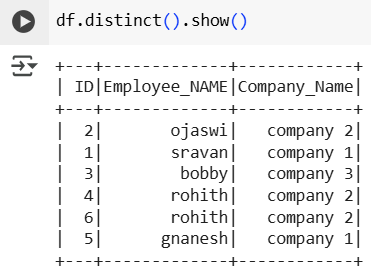
**1) dropDuplicates()**



**2) Drop duplicates based on the column name.**

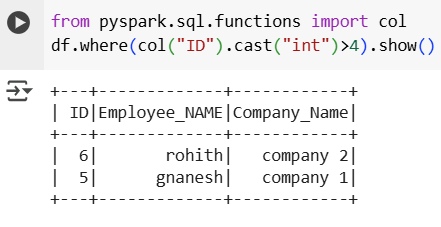
****

**Remove duplicate rows by using a distinct function**

****

###### **Dropping Rows with Condition**

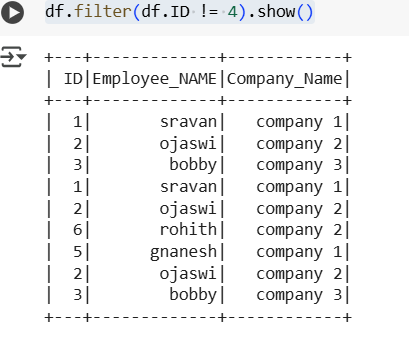
**1) Using Where condition**

****

**Not equal to**

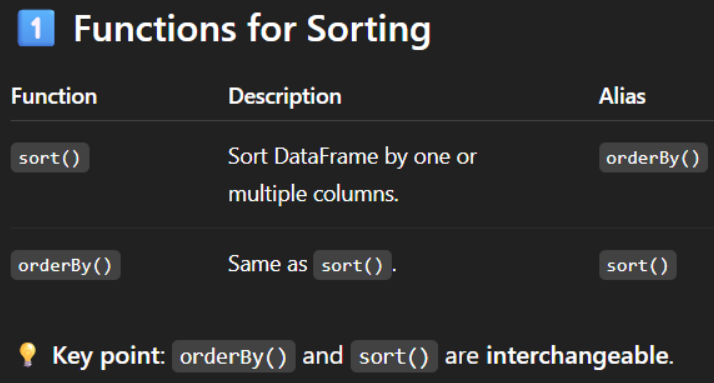
****

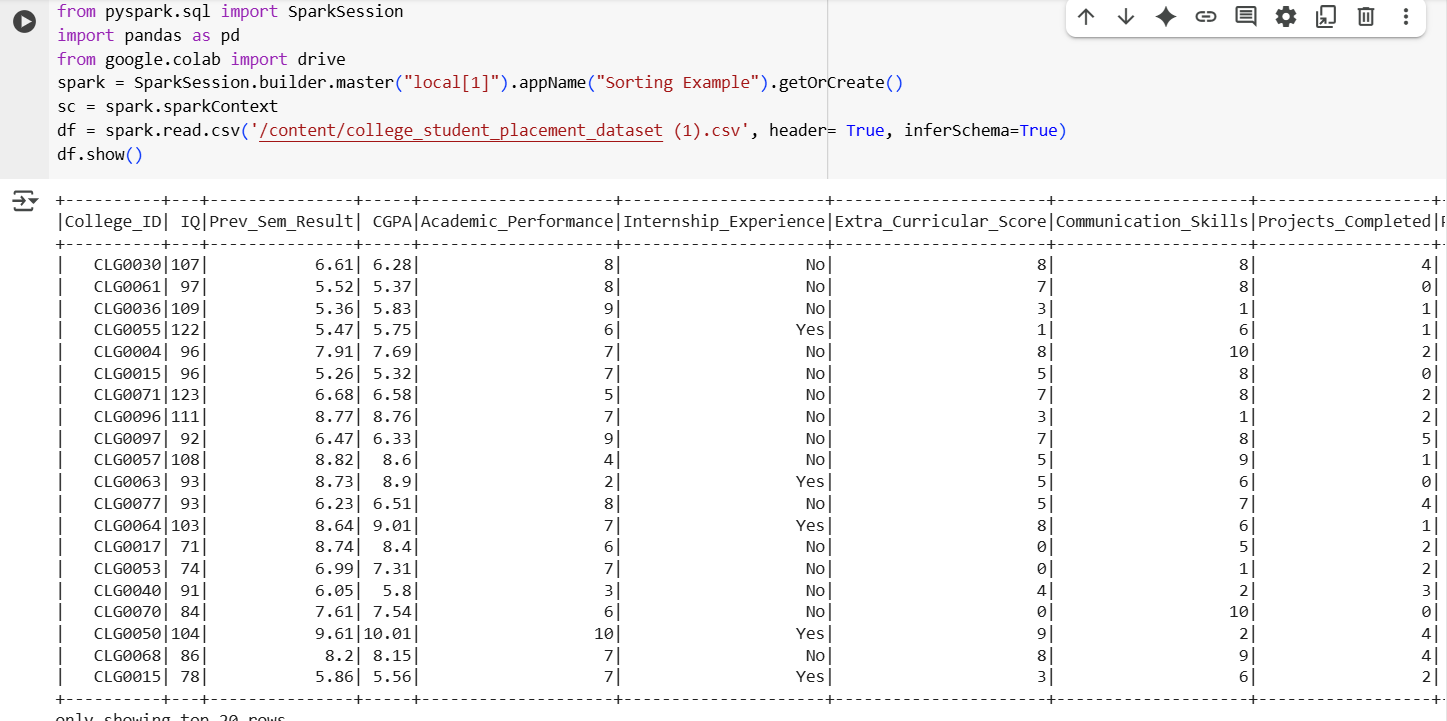
**2) Using Filter() Condition**

****

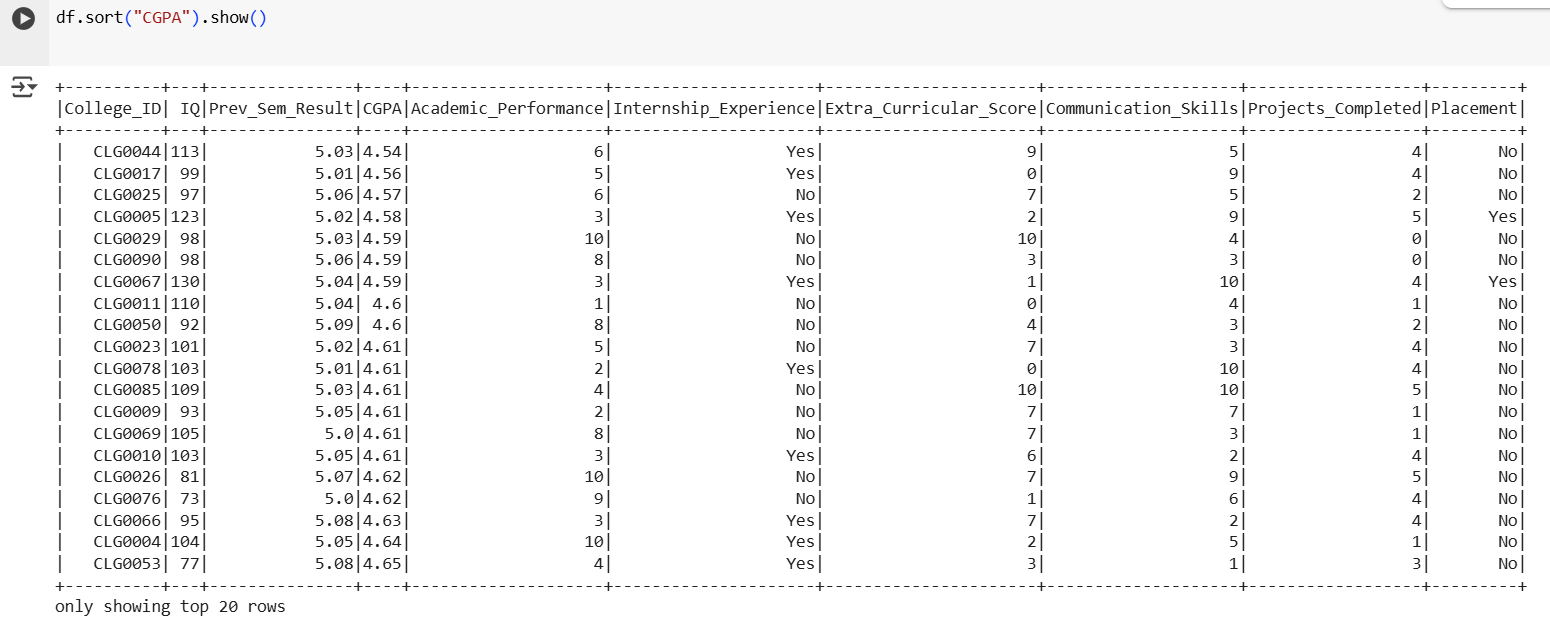
**Sorting**

Sorting in PySpark allows you to rearrange rows of a DataFrame based on one or multiple columns in ascending or descending order.

****

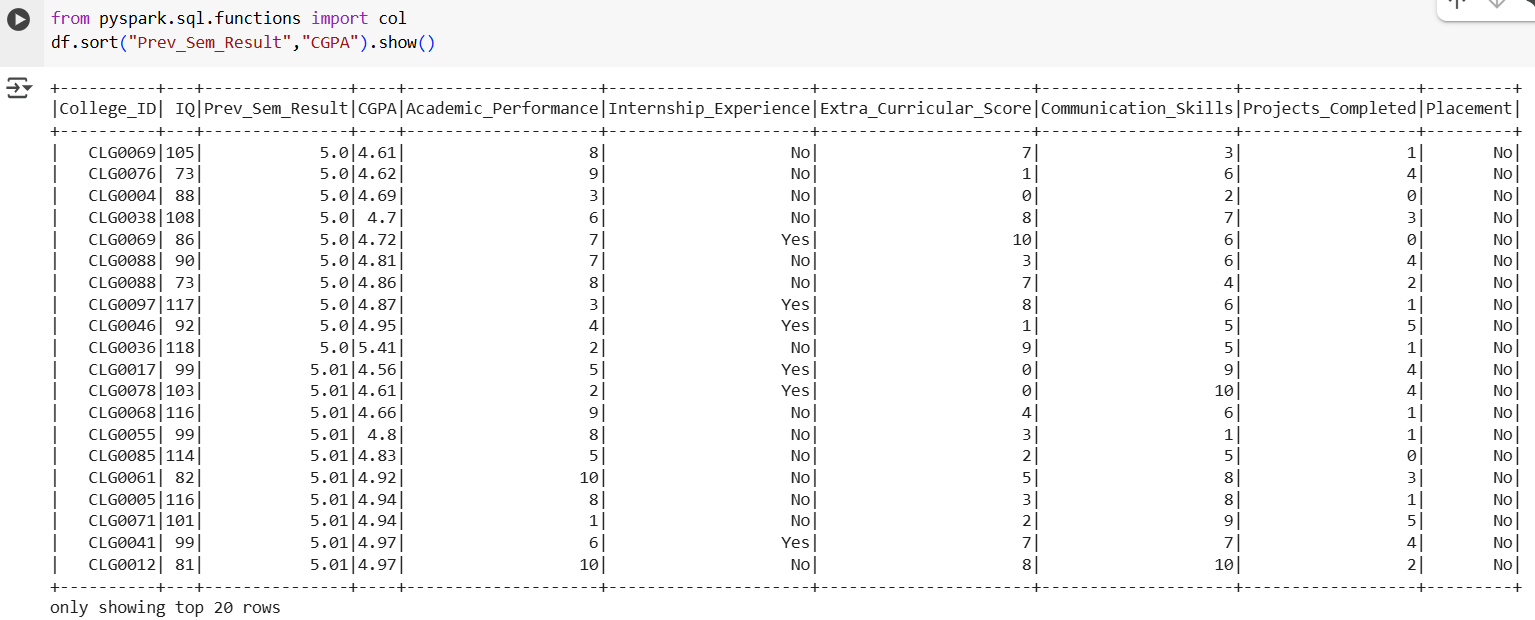


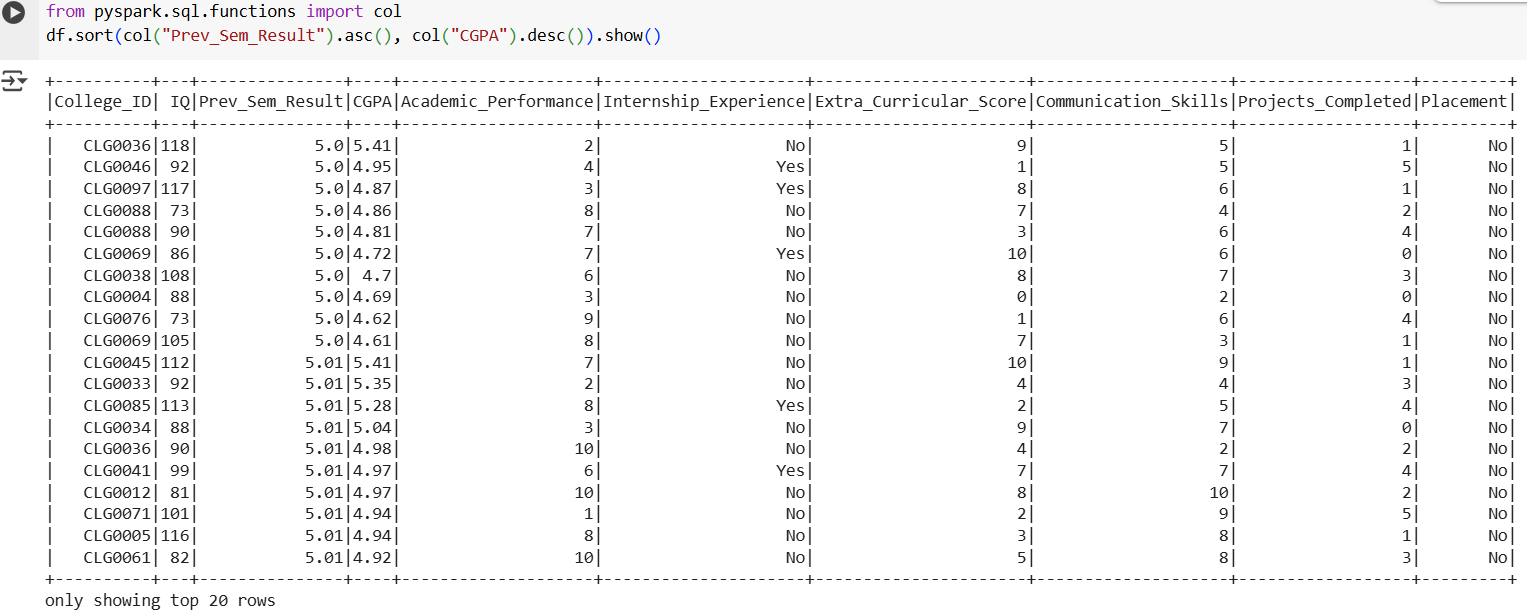
**a) Sort by Single Column (Ascending by default)**

**a) Sort by Single Column (Ascending by default)**

**b) Sort by Multiple Columns**

**a) Sort by Multiple Column**



c) Ascending & Descending Combination

## **🔍 Problem: Sorting with NULLs**

When you sort a DataFrame, PySpark needs to decide **where NULL values should go**. By default:

| **Order** | **NULL Position (Default)** |
| --- | --- |
| Ascending | Top |
| Descending | Bottom |

But sometimes, you want to **override this behavior**.

## **✅ Solution: Use These Functions for Explicit NULL Handling**

PySpark provides **4 special functions** in pyspark.sql.functions to customize NULL placement when sorting:

### **1️⃣ asc\_nulls\_first(column)**

* **Ascending sort**
* **NULLs come first**
* ✅ This is the **default behavior**

from pyspark.sql.functions import asc\_nulls\_first

df.orderBy(asc\_nulls\_first("score")).show()

**Result Example:**

+------+

|score |

+------+

| null|

| 1.2|

| 3.4|

| 4.5|

+------+

### **2️⃣ asc\_nulls\_last(column)**

* **Ascending sort**
* **NULLs go to the bottom**

from pyspark.sql.functions import asc\_nulls\_last

df.orderBy(asc\_nulls\_last("score")).show()

**Result Example:**

+------+

|score |

+------+

| 1.2|

| 3.4|

| 4.5|

| null|

+------+

### **3️⃣ desc\_nulls\_first(column)**

* **Descending sort**
* **NULLs come first**

from pyspark.sql.functions import desc\_nulls\_first

df.orderBy(desc\_nulls\_first("score")).show()

**Result Example:**

+------+

|score |

+------+

| null|

| 4.5|

| 3.4|

| 1.2|

+------+

### **4️⃣ desc\_nulls\_last(column)**

* **Descending sort**
* **NULLs go to the bottom**
* ✅ This is the **default behavior for descending**

from pyspark.sql.functions import desc\_nulls\_last

df.orderBy(desc\_nulls\_last("score")).show()

**Result Example:**

+------+

|score |

+------+

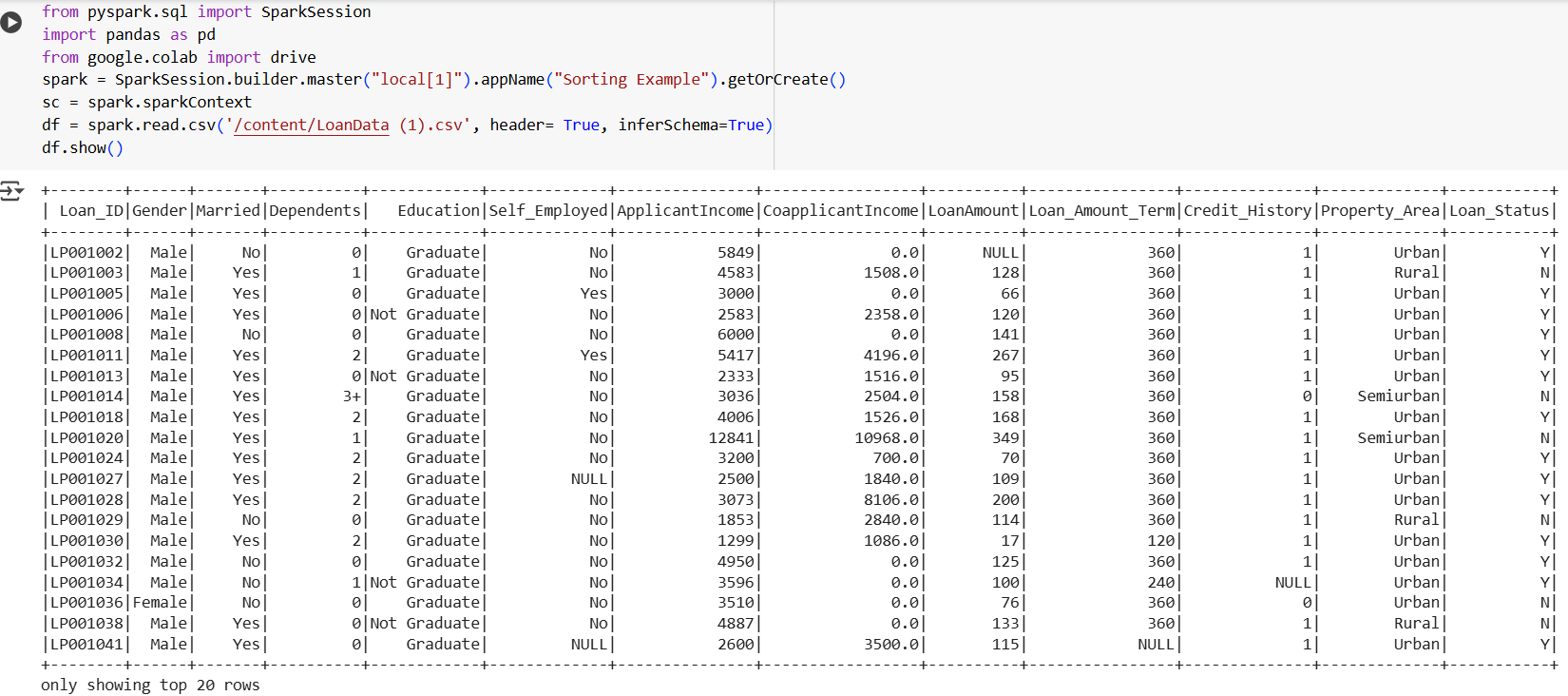
| 4.5|

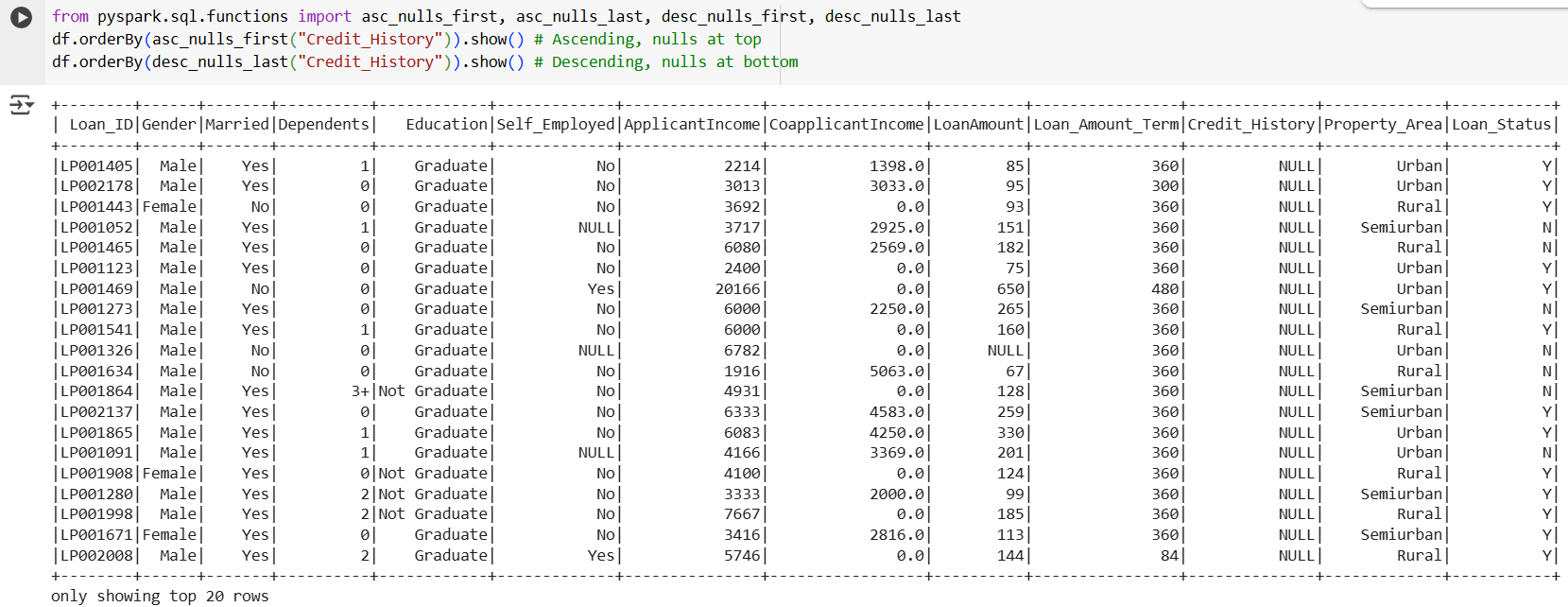
| 3.4|

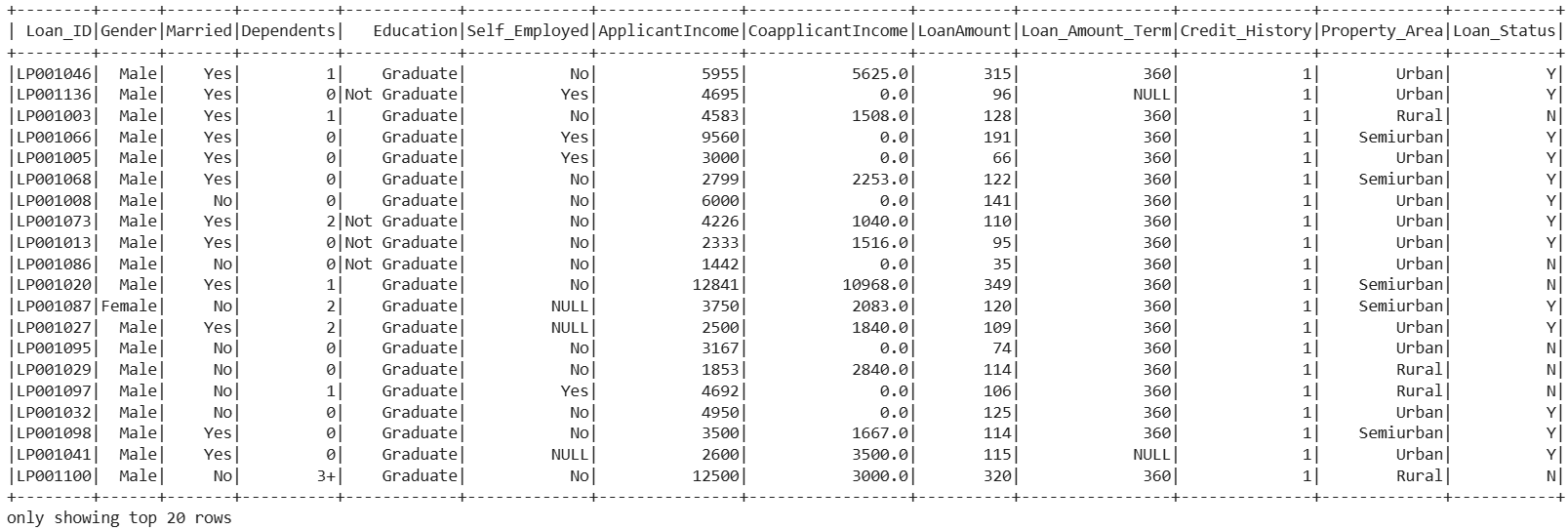
| 1.2|

| null|

+------+





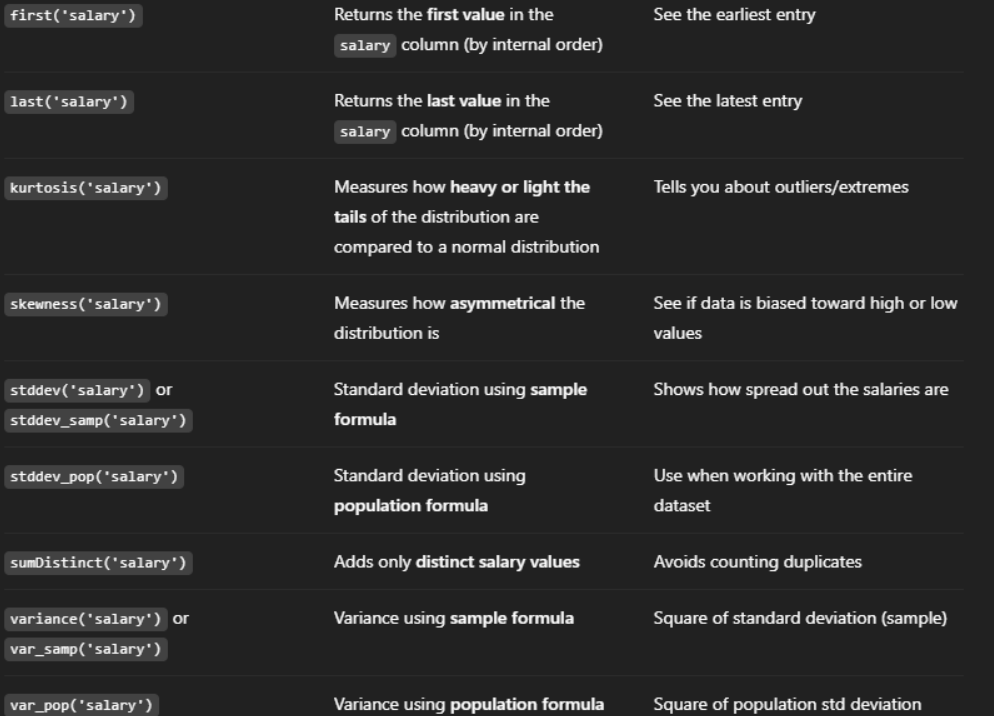


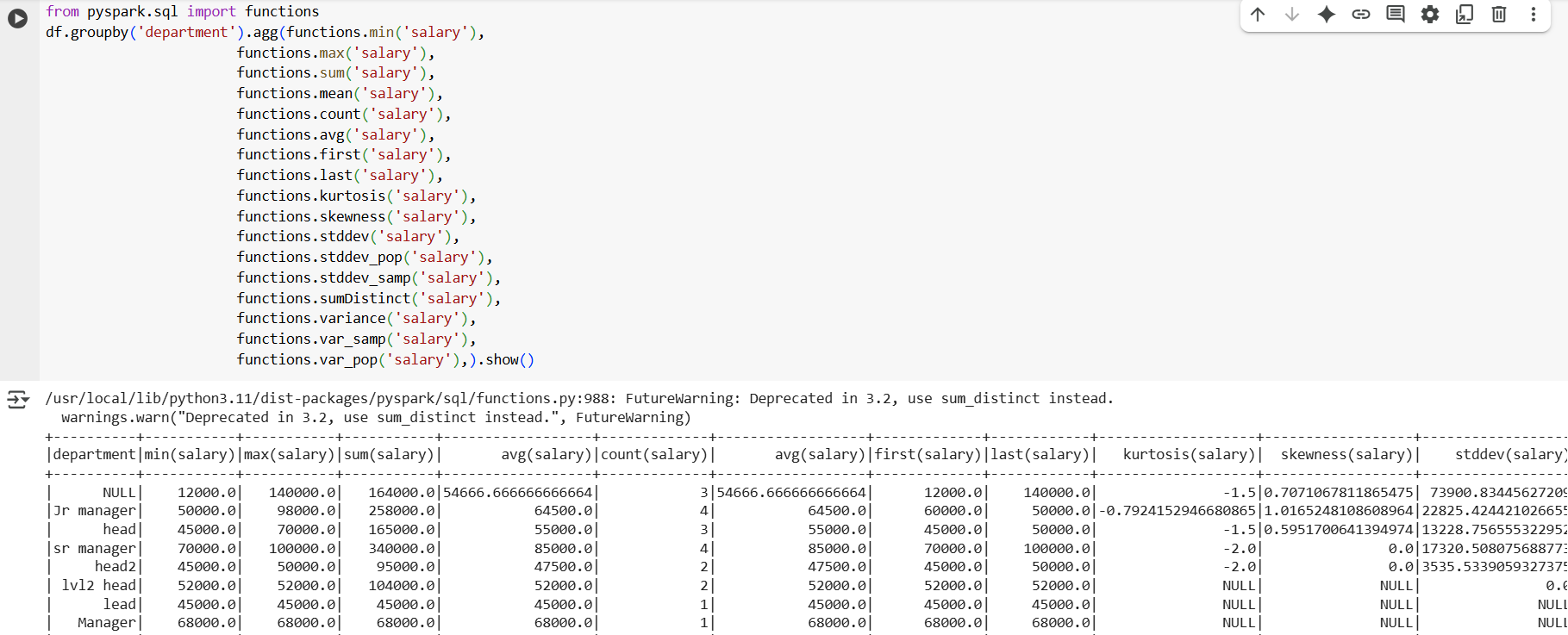
Interview/Discussion Points

* sort() and orderBy() are aliases.
* Sorting is stable (order is preserved for equal values in Spark ≥ 3.2).
* For different sorting orders on different columns, use ascending=[True, False].
* Use .asc(), .desc() for explicit ordering.
* Use asc\_nulls\_first() or desc\_nulls\_last() for null-handling.
* SQL equivalent uses ORDER BY.
* Sorting is an expensive operation because it requires shuffling data across partitions.

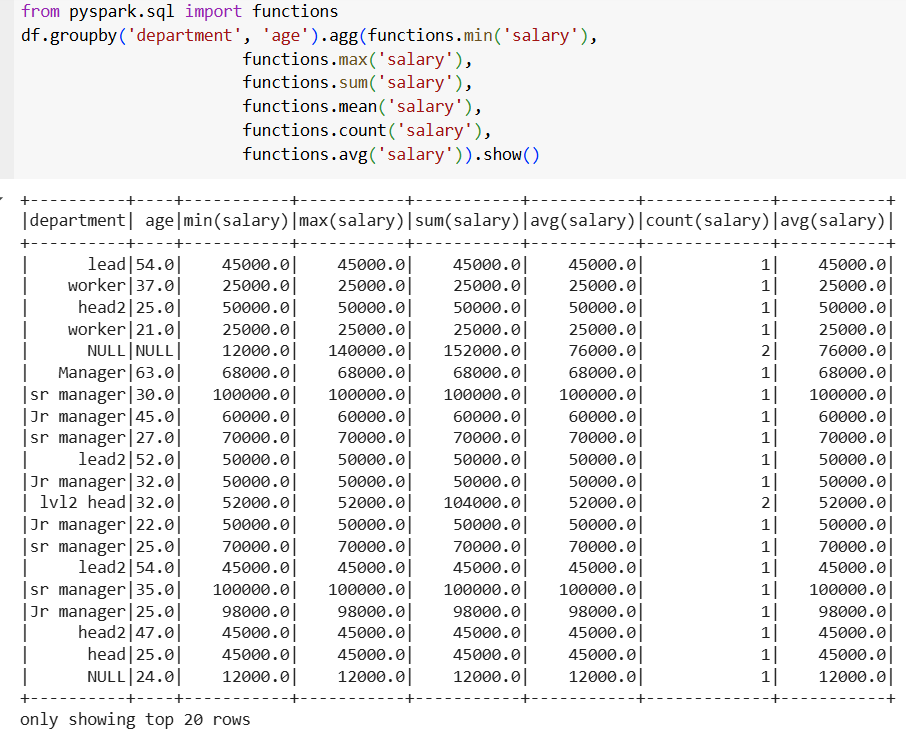
#### **Aggregate Functions**



* count(): This will return the count of rows for each group.
* mean(): This will return the mean of values for each group.
* min(): This will return the minimum of values for each group.
* sum(): This will return the total values for each group.
* avg(): This will return the average for values for each group.
* groupby() - This function groups the data by one or more columns and then applies an aggregate function to each group.

Example 1: Multiple aggregations on DEPT column with salary column

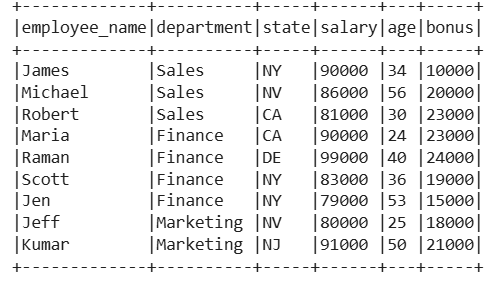
Example 2: Multiple aggregation in grouping dept and name column



## **Group By**

The groupBy() function in PySpark is used to **group data based on one or more columns**, and then perform **aggregate operations**

****

****

### **groupBy() with aggregation**

****

### **.agg() — Multiple or custom aggregations :** Use .agg() to apply one or more aggregations

### and give result column custom names using .alias().

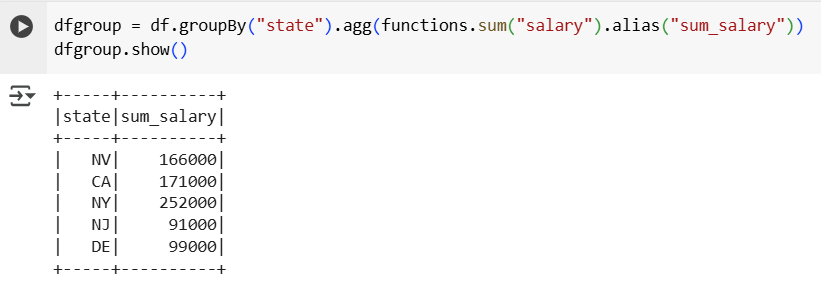
## **Syntax :**

df.groupBy("column\_name").agg(

function1(col("target\_column")).alias("custom\_name1"),

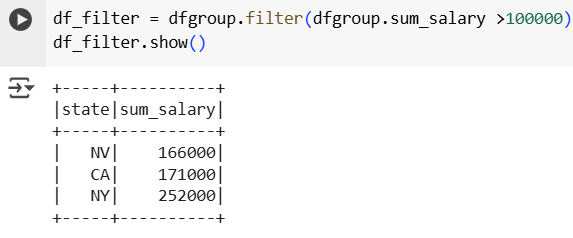
function2(col("target\_column")).alias("custom\_name2")

)



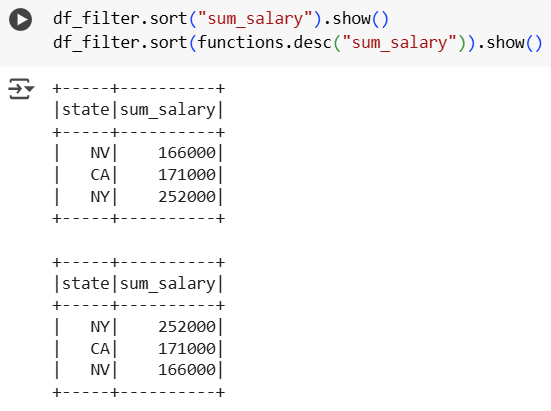
**.filter() — Filter rows after aggregation**

Filters results after aggregation, based on the aggregated value



### **.sort() — Sorting data**

Sort results in ascending or descending order.



## **Joins**

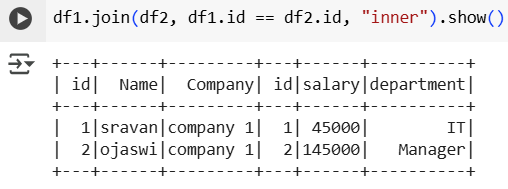


### **Inner join**

This will join the two PySpark dataframes on key columns, which are common in both dataframes.

**Syntax**

dataframe1.join(dataframe2,dataframe1.column\_name == dataframe2.column\_name,"inner")

****

### **Full Outer Join**

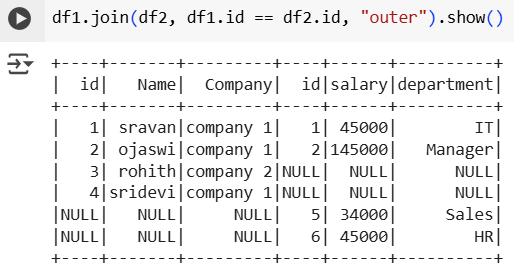
A FULL OUTER JOIN combines all the data from both tables (DataFrames).If there is a match between the two DataFrames, it joins the rows.If there’s no match, it still keeps the row and fills the missing values with null., we can perform this join in three ways.They are just different words for the same join , does same work just name is different

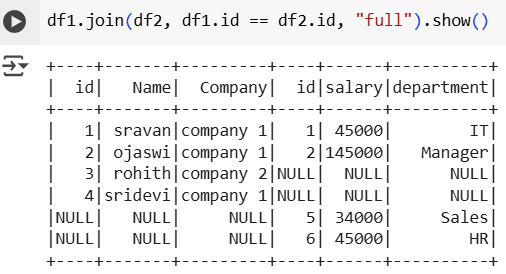
**Syntax**

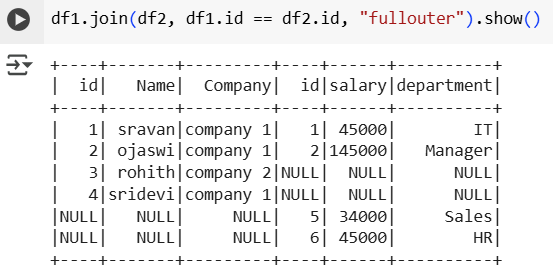
**outer:** dataframe1.join(dataframe2,dataframe1.column\_name == dataframe2.column\_name,"outer")

**full:** dataframe1.join(dataframe2,dataframe1.column\_name == dataframe2.column\_name,"full")

**fullouter**: dataframe1.join(dataframe2,dataframe1.column\_name == dataframe2.column\_name,"fullouter")







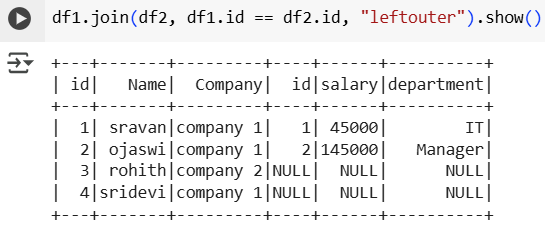
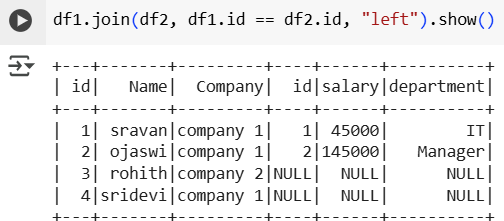
### **Left Join**

* Returns all rows from the left DataFrame and the matched rows from the right DataFrame.
* If no match is found in the right DataFrame, NULL values will be returned for the right DataFrame columns.

**Syntax**

df1.join(df2, df1.ID == df2.ID, "left")

df1.join(df2, df1.ID == df2.ID, "leftouter") # same as left



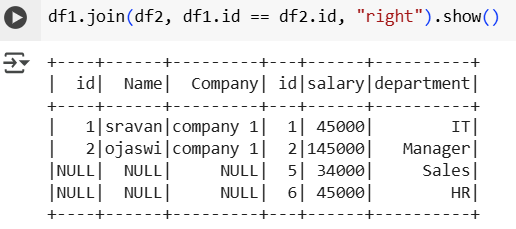
### **Right Join**

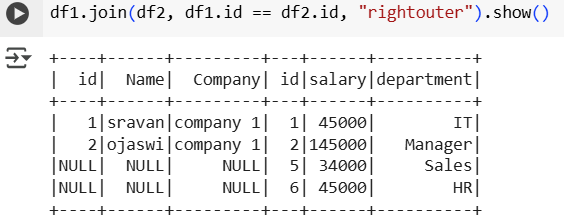
* Returns all rows from the right DataFrame and the matched rows from the left DataFrame.
* If no match is found in the left DataFrame, NULL values will be returned for the left DataFrame columns.

**Syntax**

**right:** dataframe1.join(dataframe2,dataframe1.column\_name == dataframe2.column\_name,"right")

**rightouter**: dataframe1.join(dataframe2,dataframe1.column\_name == dataframe2.column\_name,"rightouter")



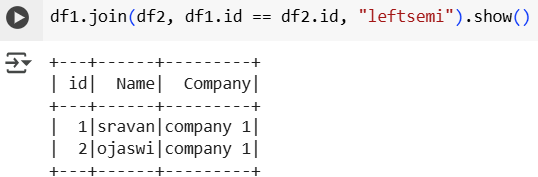


**Leftsemi join**

* Returns only the rows from the left DataFrame that have a matching key in the right DataFrame.
* Unlike left join, it does not return any columns from the right DataFrame.

**Syntax**

dataframe1.join(dataframe2,dataframe1.column\_name == dataframe2.column\_name,"leftsemi")

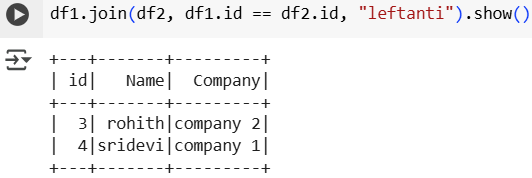


### **LeftAnti join**

* Returns only the rows from the left DataFrame that do NOT have a match in the right DataFrame.
* It's like filtering out all matches and keeping only the unmatched records.

**Syntax**

dataframe1.join(dataframe2,dataframe1.column\_name == dataframe2.column\_name,"leftanti")



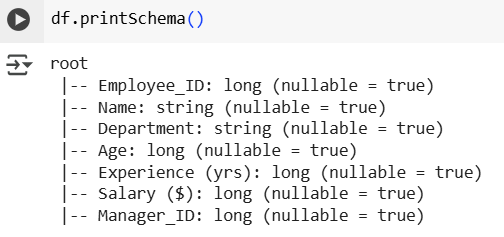
## **Handling Null Values In Pyspark**

Handling missing values (NULL/NA) is one of the most critical steps in data preprocessing. PySpark provides multiple tools to detect, drop, or fill missing values efficiently, especially for big data.



### **Detecting Missing Values**

If nullable=True for a column → It contains missing values (or it can potentially contain nulls).

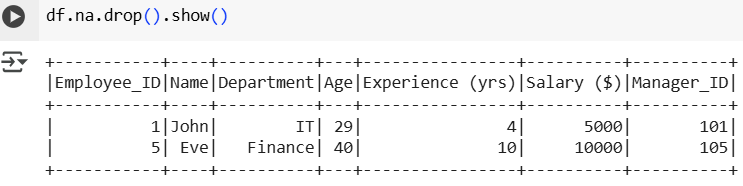


### **Dropping NULL Value**

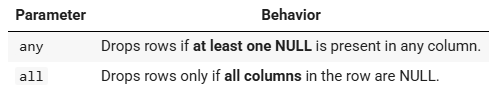
PySpark provides df.na.drop() to remove rows with missing values.

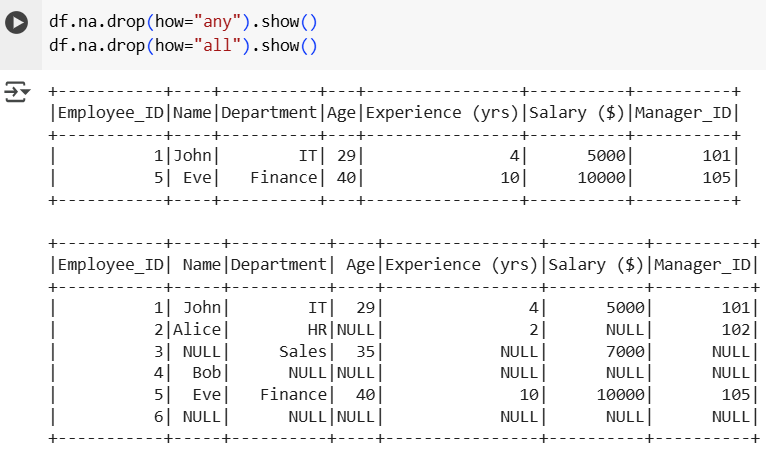
#### **a) Drop all rows with any NULL value**

Removes rows if any column has a NULL value.



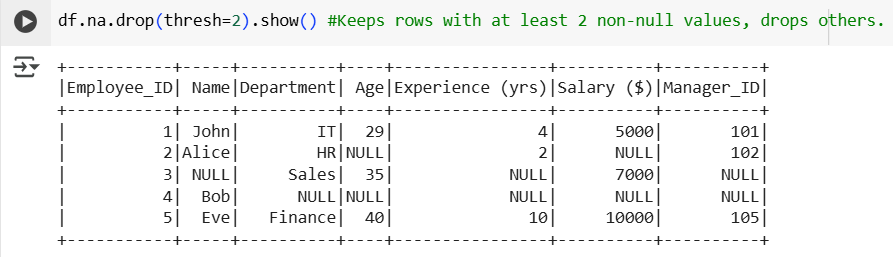
#### **b) Using how parameter :** Controls when to drop rows



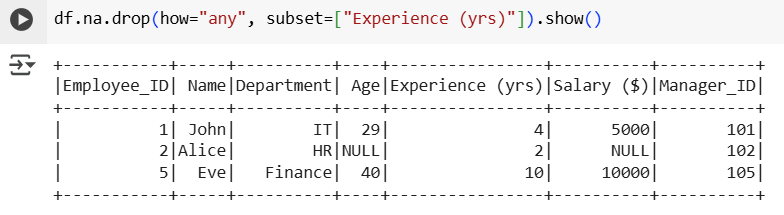


#### **c) Using thresh parameter**

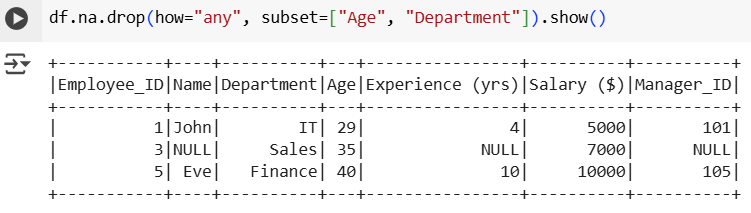
Drops rows based on a minimum number of non-null values required:



#### **d) Using subset parameter :** Drop null values only for specific columns



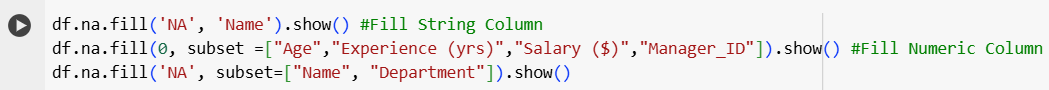
**Passing multiple columns to drop the null values**

****

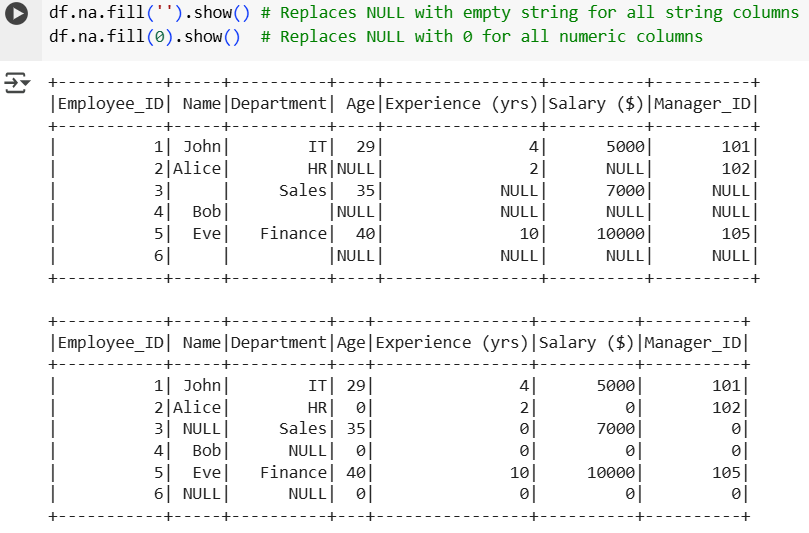
### **Filling Missing Values (Imputation)**

Instead of dropping rows, we can replace NULL values using fill() or fillna().

#### **a) Fill NULL values with a constant**



#### **b) Fill NULL values for all string or numeric columns**

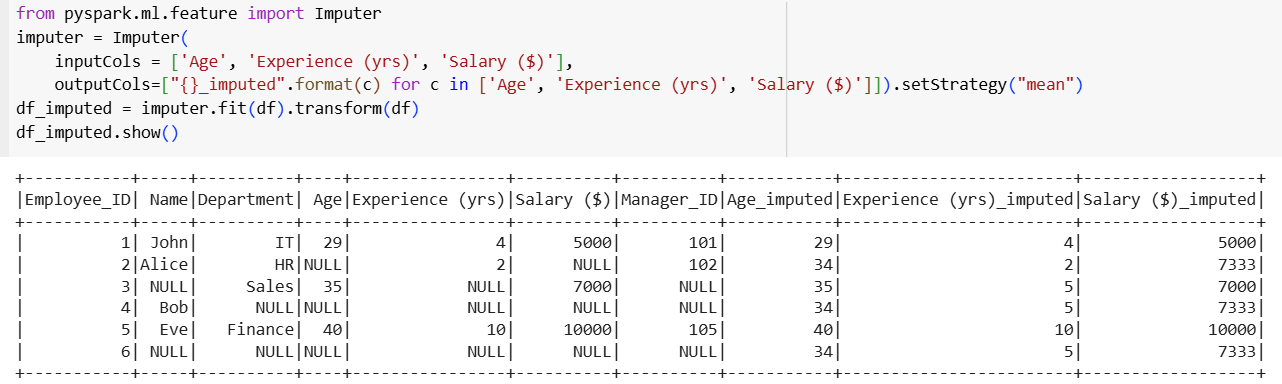


#### **c) Fill NULL values using a dictionary (different values for different columns)**

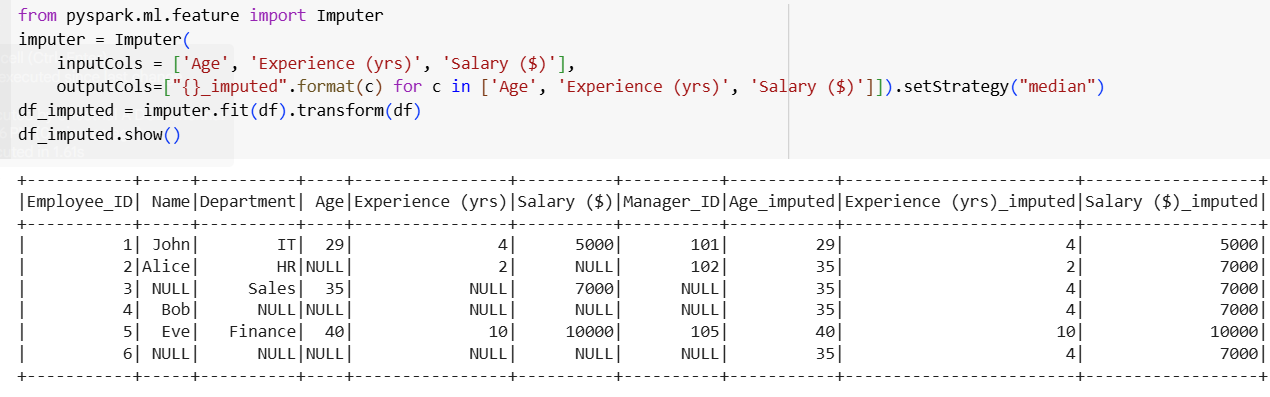


### **Imputing Missing Values using Statistics (Mean, Median, Mode)**

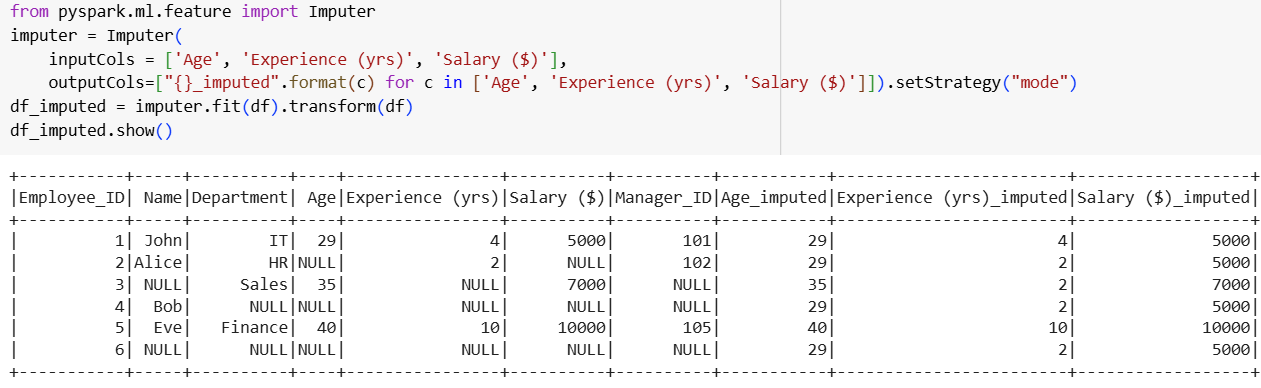
PySpark's Imputer (from pyspark.ml.feature) can replace NULL values with the mean, median, or mode.

**Mean**

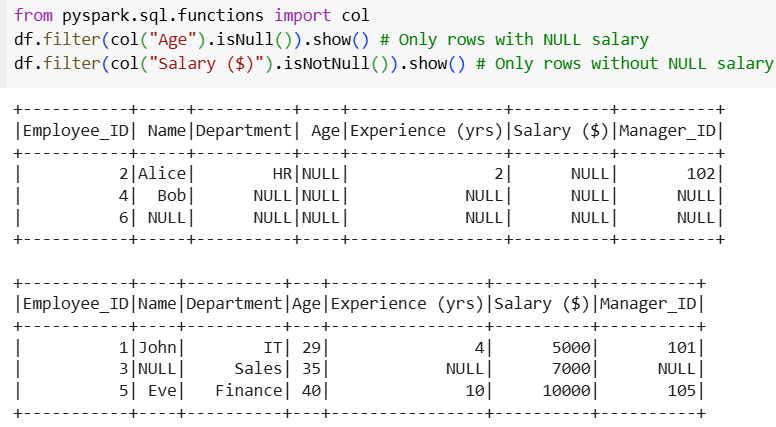
**Median**

****

**Mode**

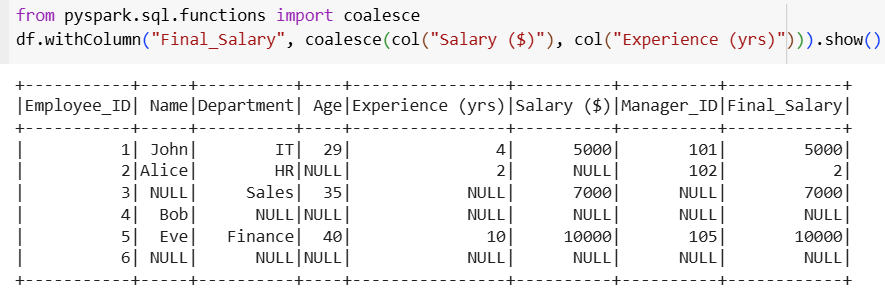
****

### **Filtering NULL Values**

Instead of dropping or filling, you can filter rows using isNull() or isNotNull():

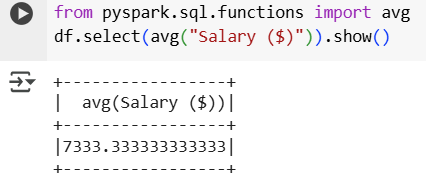
### **Using Coalesce() for NULL Handling**

coalesce() returns the first non-null value among columns



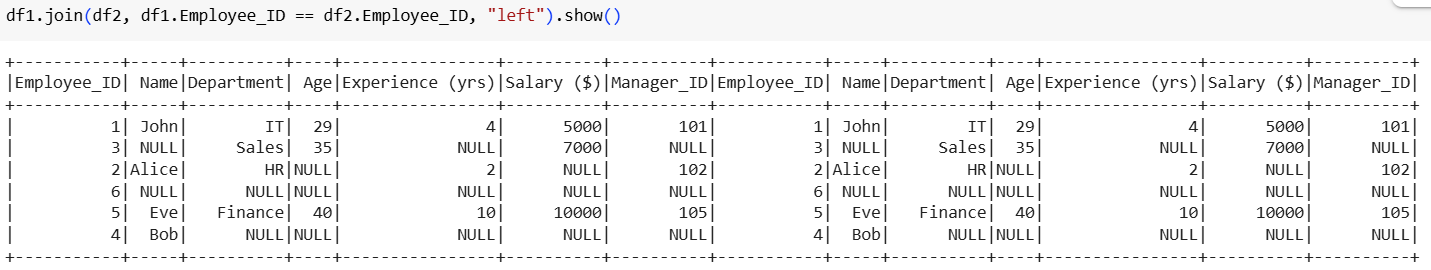
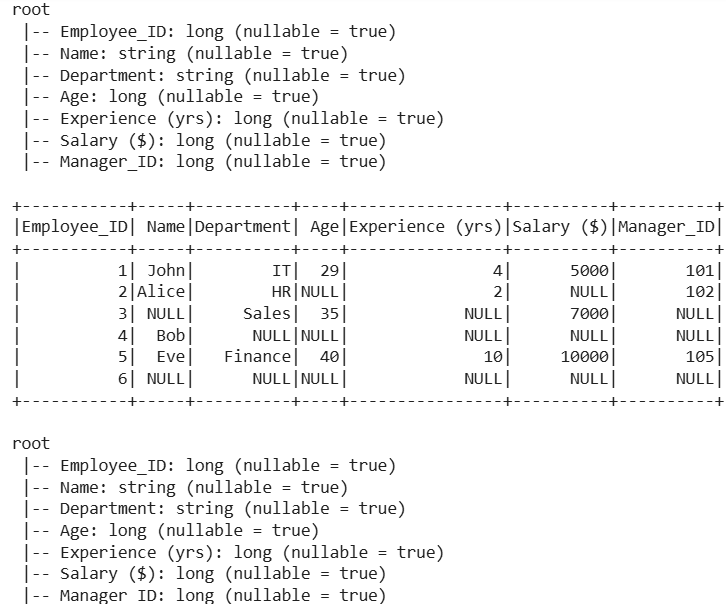
### **Aggregations Ignore NULLs**

When performing aggregations (avg, sum, etc.), Spark ignores NULL values:



### **NULL Handling in Joins**

When joining two DataFrames, NULLs in join keys do NOT match unless explicitly handled.

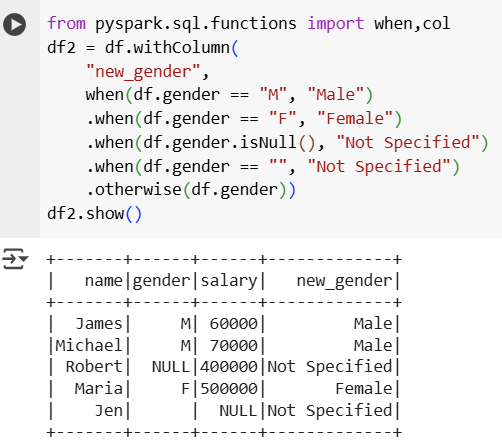


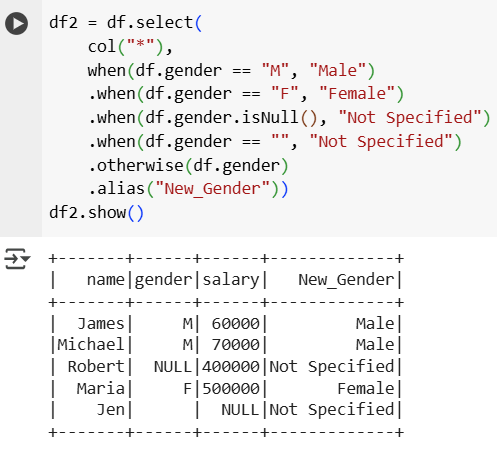
If ID is NULL in either DataFrame, it will not match. Use fillna() or coalesce() before joining if required.

## **When - Otherwise**

* when is a conditional function from pyspark.sql.functions.
* It is used to apply "if-else logic" (similar to SQL CASE WHEN) to transform or create new columns.
* Always used with .otherwise() for the "else" part (if no conditions match).



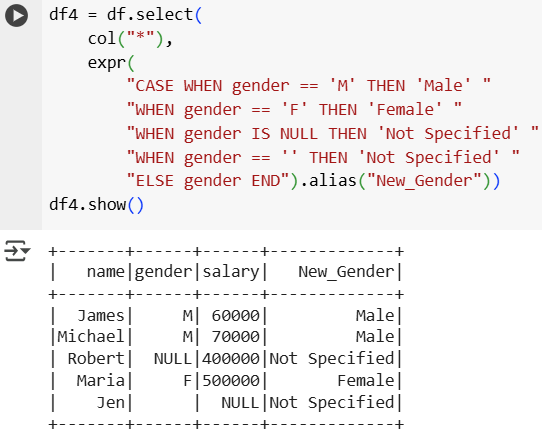


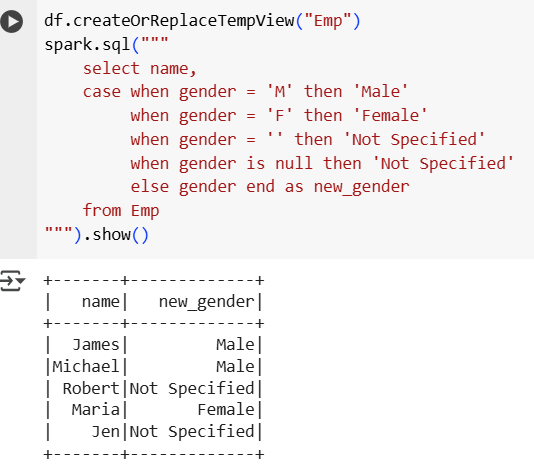
**Alternative Method: Adding New Column with select**

**Using SQL CASE WHEN with expr**

****

**Using select with SQL Expression**

****

**Using Spark SQL Directly**

## **Union() and UnionALL()**

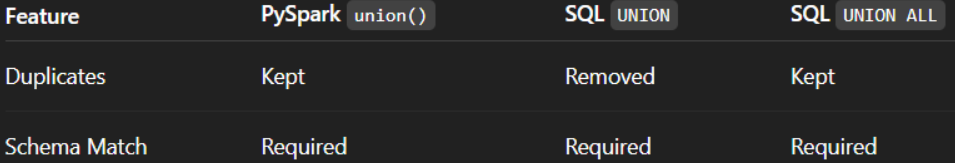
**What is union() in PySpark?**

* union() is a transformation used to merge two or more DataFrames.
* Both DataFrames must have the same schema (same column names and data types).
* It returns all rows from both DataFrames, including duplicates.
* If you want only unique rows, you must call .distinct() after union().

**What is unionAll()?**

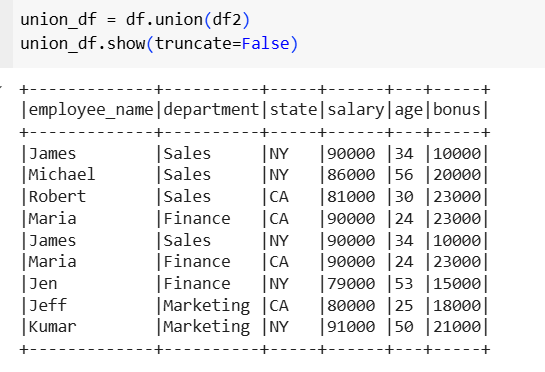
* unionAll() was used in older Spark versions (before 2.0.0).
* It is deprecated and replaced with union().
* In PySpark, union() and unionAll() behave the same: both keep duplicates.
* If you come across unionAll() in legacy code, you should replace it with union().

**Difference Between PySpark and SQL Union**

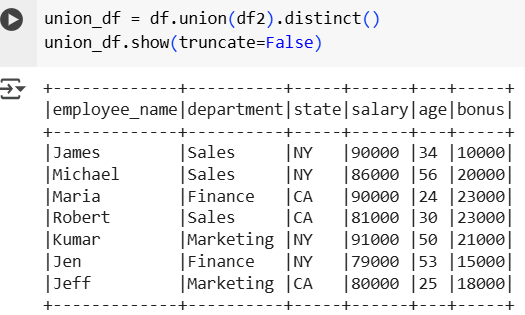
****

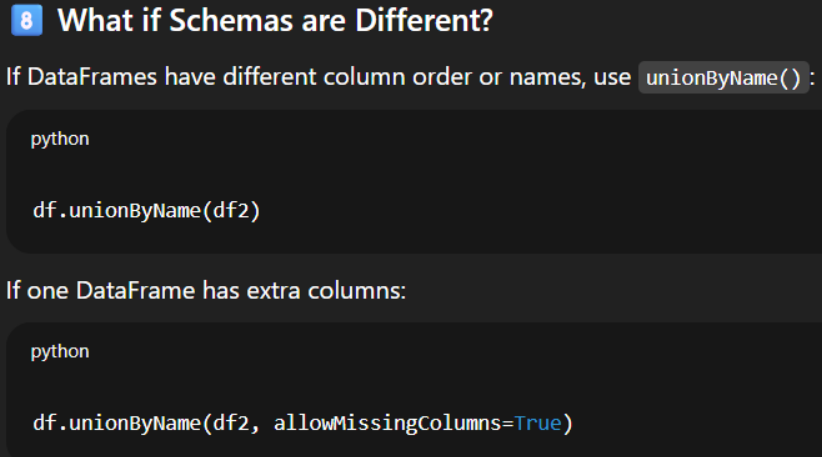
#### **Creating Two DataFrames**



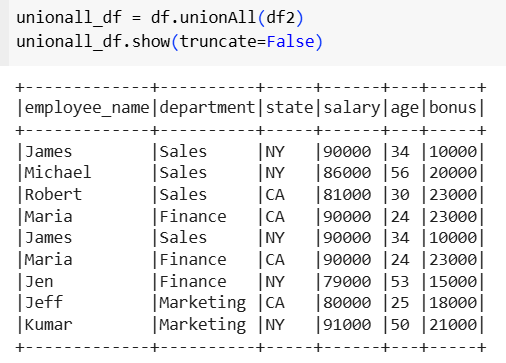


**Removing Duplicates in union**

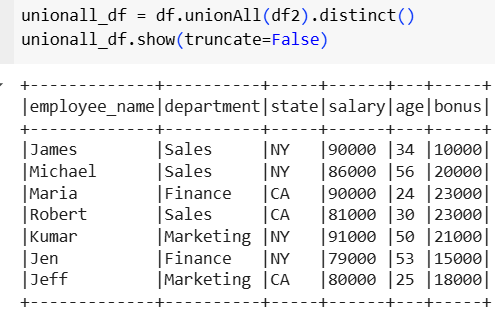
****

****

#### **Union All Example**

****

**Removing Duplicates in unionAll()**

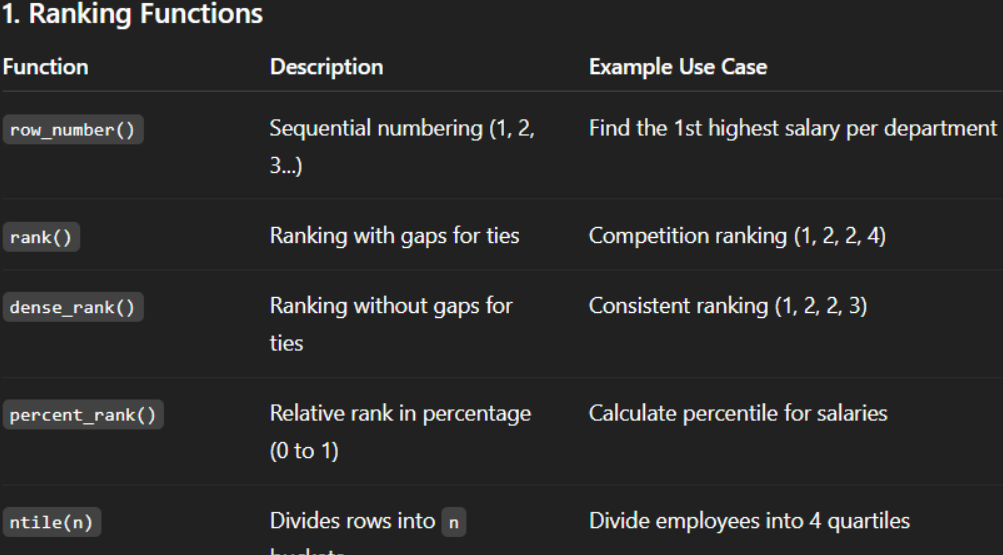
****

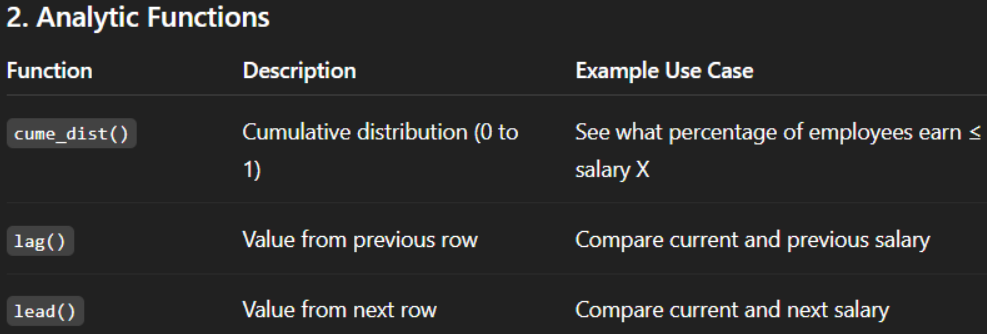
**Window Functions**

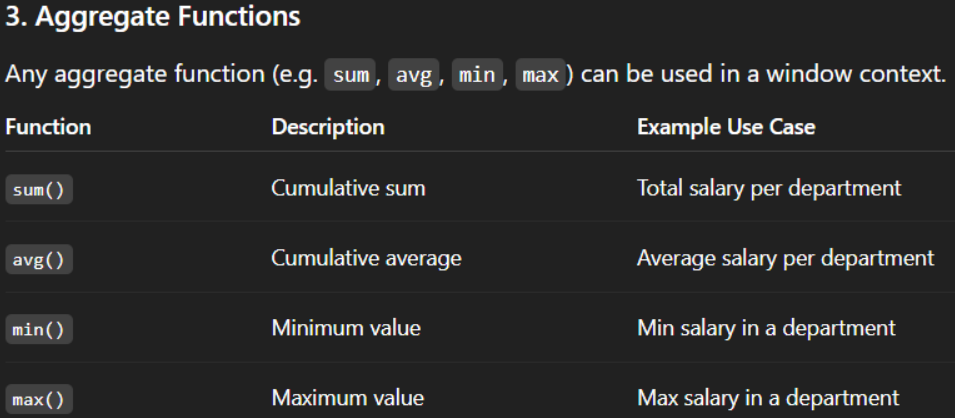
Window functions let you:

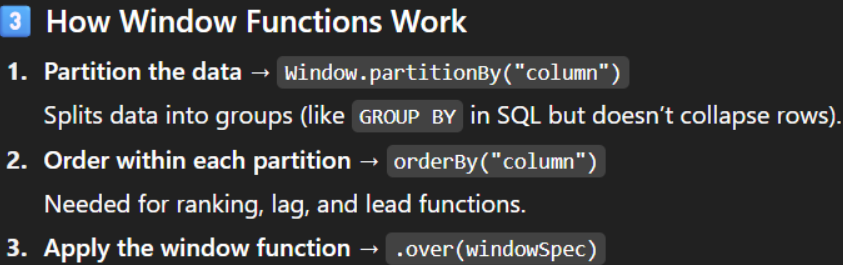
* Look at neighboring rows (before/after).
* Perform aggregations without grouping into fewer rows.
* Use row numbers, ranks, and lag/lead functions across a defined "window" of data.

### Types of Window Functions



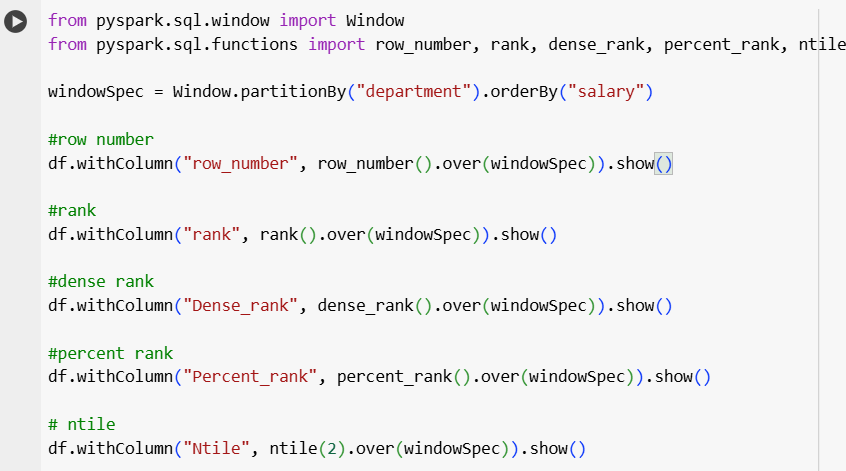


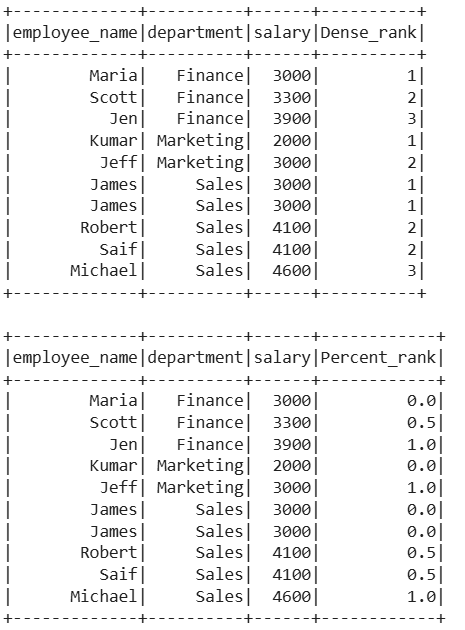
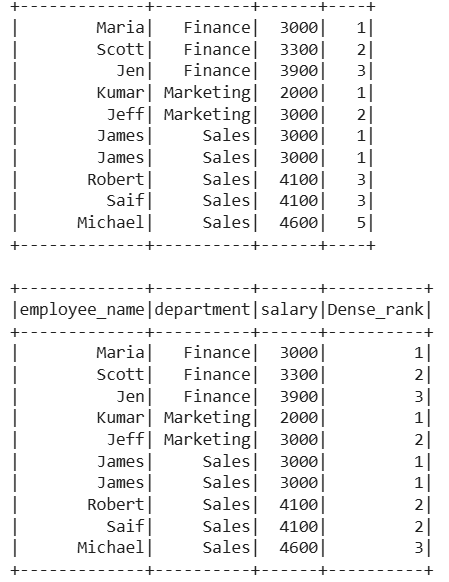


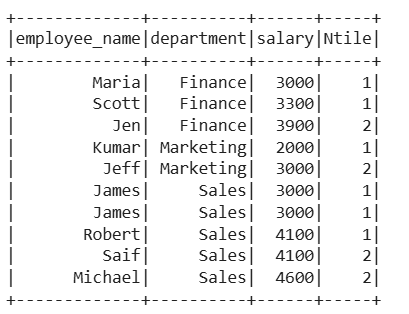




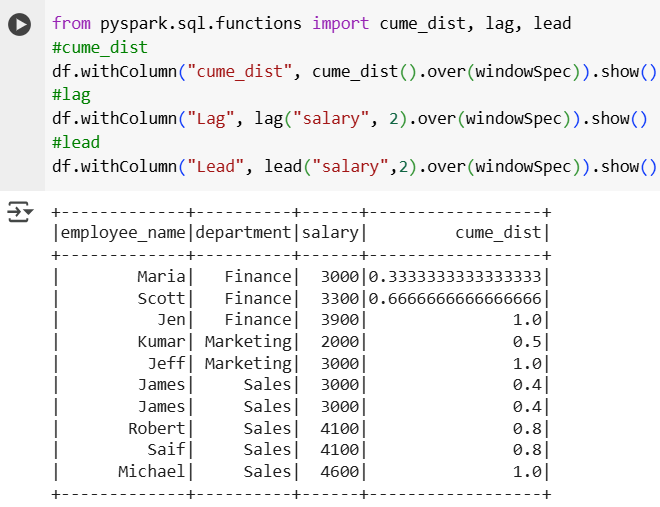
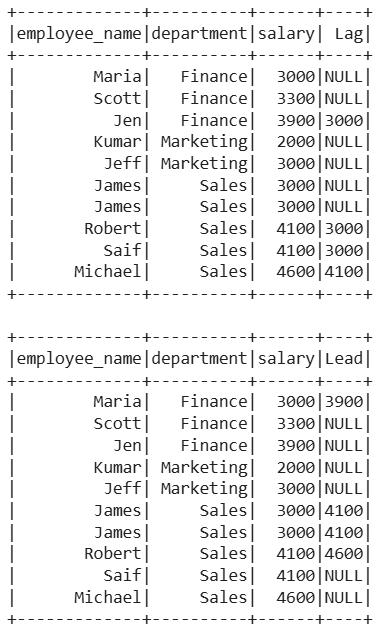
#### Ranking Functions

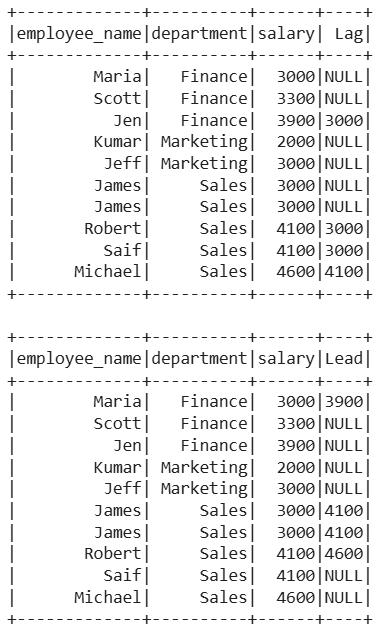




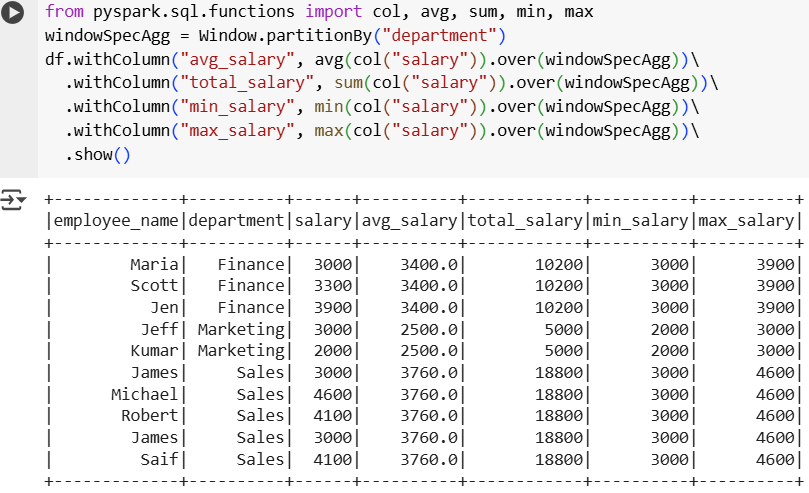
****

#### Analytic Functions

****

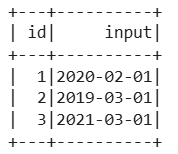
****

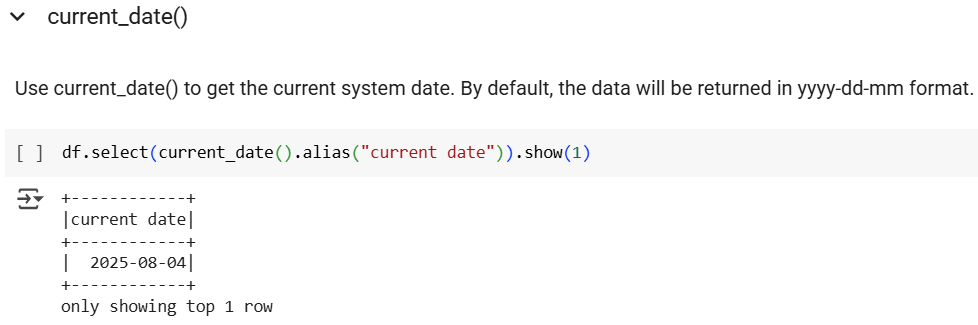
#### Aggregate Functions

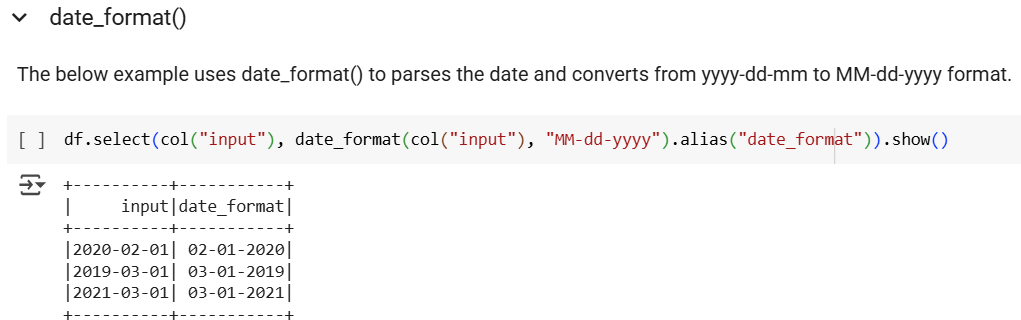
****

## Date and Time Functions

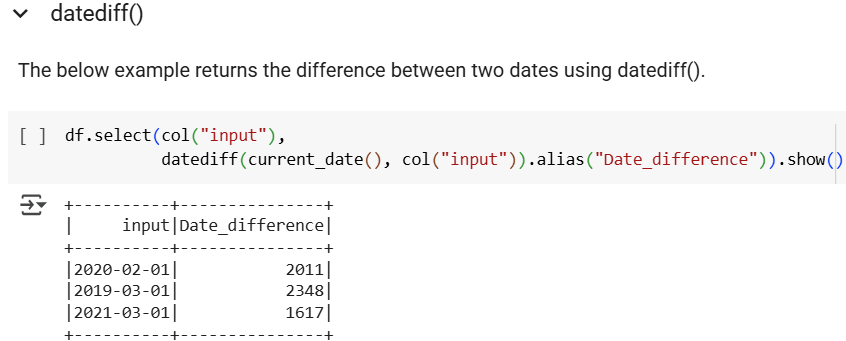


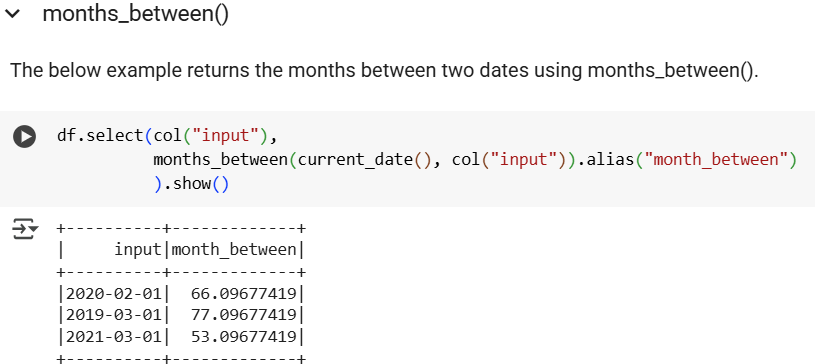


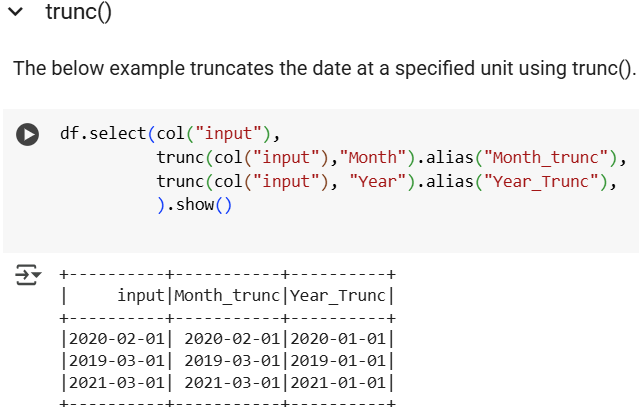


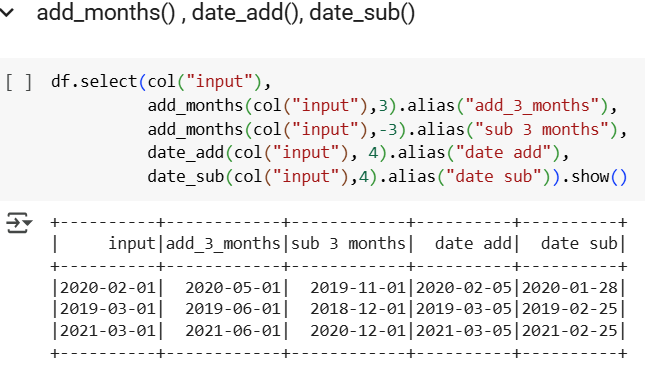
****

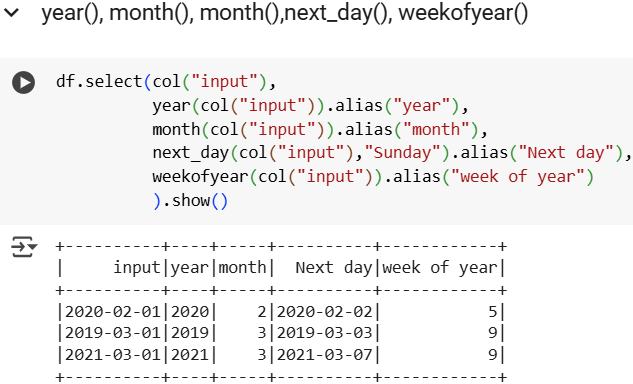
****

****

****

****

****

****