**Centralized System**

In centralized system data is stored on one server or system and processed on single server.

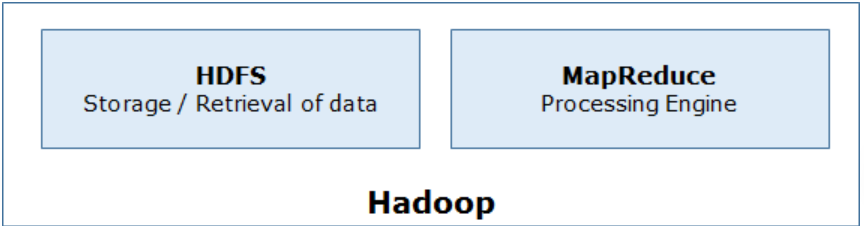
Now the problem is that when data increases then we need to upgrade server to store and process more data but even by upgrade we will come to a point when it’s not possible to upgrade server any more to handle there we need distributed system.

**Distributed System:**

In Distributed system huge amount of data is split across multiple systems or servers, and processed in a distributed manner. This is called distributed computing and Apache Spark is one of the open-source frameworks for distributed computing

|  |  |
| --- | --- |
| Distributed System | Centralized system |
| Data is stored on multiple system. | Data is stored on single system/Server |

**Ecosystem of Hadoop:**



Hadoop is made up two main parts -

HDFS - for storage and retrieval of data in a distributed system

MapReduce - for processing the data present in the distributed system.

In Hadoop, MapReduce is the processing framework. So, if **we were to write a program to compute something the processing would be done using MapReduce**

**In Pyspark computing in done in memory instead of disk.**

Sprak Core Process or Execution:

Spark is meant for high-speed computing on distributed systems. In a distributed system, the data is distributed across nodes (systems in a cluster), and any operation submitted to Spark is executed in parallel across these nodes. So, there should a **supervisor, who monitors which node is preforming which operation. This supervisor is called 'Driver'** in Spark, and **each node that perform the assigned task is called 'Executor'**.

**MapReduce**

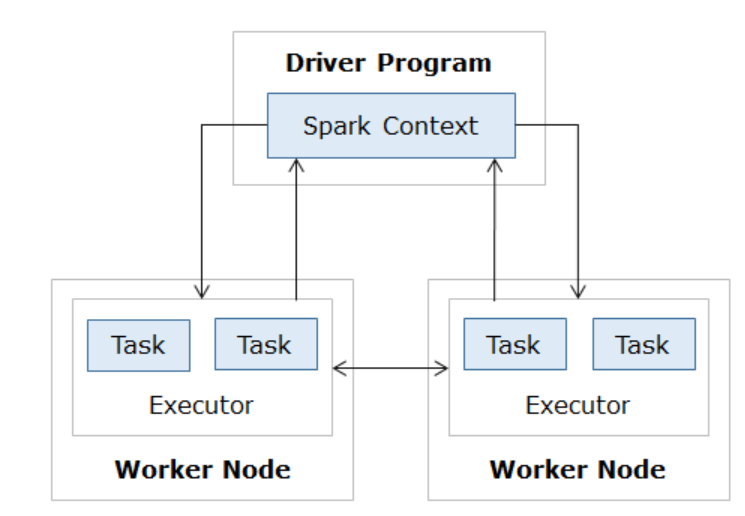
In MapReduce, data is read from disk to operation then write back to disk for each operation.

So, in a program if we are doing 10 operations then this sequence will be done 10 times.

**Spark:**

In spark data is stored in RAM or memory and operation is perfomed in memory that will save time form reading and writing back to disk leading high speed.

Assume that we are supposed to find the average marks of students for each subject. So, when the job is submitted to Spark, it immediately allocates a supervisor (Driver) to over look the job submitted. This supervisor gives instructions to the Executors about the 'Task' that they are supposed to complete. 'Driver' program performs its various operation by communicating to Spark through a 'SparkContext' object. SparkContext is the starting point of any spark application. So, if we were to write a Spark Application to find the sum of marks of students, the first objective would be to create an object of SparkContext



**Spark Execution Architecture:**

|  |  |
| --- | --- |
| In more detail level |  |

|  |  |
| --- | --- |
| **In short** |  |

**1) Driver Program**

Driver program is the **central coordinator** of a Spark application. It contains the SparkContext, which is the entry point to Spark's execution environment. The driver program:

* **Splits your application into tasks**
* **Schedules tasks on executors**
* **Coordinates the overall execution of your Spark job**

**2) Cluster manager**

Allocates resources (such as CPU and memory) across the cluster. Spark can run on various cluster managers, including:

* **Standalone** — Spark’s built-in cluster manager.
* [**YARN**](https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html?ref=chaosgenius.io) — The resource management layer for Hadoop.
* [**Apache Mesos**](https://mesos.apache.org/?ref=chaosgenius.io) — A general-purpose cluster manager.
* [**Kubernetes**](https://kubernetes.io/?ref=chaosgenius.io) — For deploying Spark in containerized environments.

The cluster manager decides which worker nodes your application gets to use.

**3) Executors**

Executors are worker processes that run on worker nodes. Each executor is responsible for executing the tasks (unit of work that will be sent to one executor) assigned to it by the driver and reporting the results back to the driver. Executors also store data in memory or on disk, depending on the application's requirements.

**4) Worker Nodes**

Worker nodes are the machines in the cluster where executors run. Each worker node can host multiple executors, and the tasks are distributed among these executors for parallel processing.

**Job**

* The highest-level abstraction in Spark, representing a complete computation.

**Stage**

* A *Stage is a collection of tasks that share the same shuffle dependencies*, meaning that they must exchange data(if needed) with one another during execution. *Each stage is composed of one or more tasks.*
* *Stages are executed sequentially, with the output of one stage becoming the input to the next stage.*
* By breaking down a job into stages, Spark can schedule tasks in a way that maximizes parallelism while minimizing the amount of data that needs to be shuffled between nodes.
* Stages are separated by operations that require data shuffling, such as groupBy, reduceByKey, and joins.

**Task**

* A task is the smallest unit of work in Spark. It represents a single computation that is executed on a partition of the data.

**Key Differences:**

* **Task**: Refers to the smallest unit of computation, typically processing one partition of the data.
* **Stage**: Refers to a set of tasks that can be executed in parallel without requiring a shuffle.

**API in PySpark**

Apache Spark provide three type of APIs

1. RDD
2. DataFrame
3. Dataset



**RDD: Resilient Distributed data**

**Distributed:** --- Stored on multiple nodes/servers.

**Resilient:**

**Fault Tolerence** – RDD track data lineage information to recover lost data quickly and automatically on failure at any point of execution cycle.

We can see from below picture that at every transformation on any RDD a new RDD is created and a lineage/detail is maintained for each RDD



**Executing Applications on Spark**

Once spark services have been started and are in working state, we can execute spark application in two ways.

1. Spark Interactive Shell
2. Execute Spark Script

**Executing in Spark Interactive Shell**

For executing in spark interactive shell open the **command window and type 'pyspark'**, this will open spark interactive shell. It has **default session context created by name 'sc'.**

Using the session context now we can perform out action.

**Executing in Spark Script:**

This is line creating a .py file and running that python script file.

**Pyspark vs MapReduce or Haddop**

Several optimizations on Saprk have made it faster than MapReduce here we shall discuss two main reasons –

1. in-memory execution
2. lazy evaluation

**# In-Memory execution #**

Assume that there are 10 operations to be performed on data in sequence. Let us see how MapReduce and Spark execute this operation-

**Map Reduce**

* Reads data from disk
* Executes the first operation on the data
* Writes the result back to disk
* Reads the result of the first operation from disk
* Executes the second operation on the data
* Writes the result of the second operation back to disk, and so on.

**Spark**

In case of Spark, the **result of the previous execution is stored in-memory and not on disk**. So, Spark saved a whole load of time by avoiding disk-IO operations

**# Lazy Evaluation#**

Lazy evaluation means the execution will not start until an action is triggered or will trigger only on those data which is required for action.

Spark creates a sequence of operation or transformations to be performed on the data called a directed acyclic graph (DAG) and then evaluated.

Let’s take below example-

**Transformation will be executed only when action is called.**

**CAE 1:** Let say we have to perform below operation in a script

1. import dataset
2. retain only the first 10 rows
3. split the data based on comma delimiter
4. aggregate the data

Here first three statement/operation are transformation and 4th statement is action.

In this case, the first three steps (which are 'transformations') will get executed only during the execution of the fourth command (which is an 'action')

**CASE 2**: We have to read data from file and print first 5 lines. File size is 1TB.

**MapReduce:**

It will read/import 1TB file and then will print first 5 line.

**Spark:**

In spark it will not import/read blindly 1TB file but will import/read only first 5 lines and print them.

**Pyspark vs MapReduce(Hadoop)**

| Feature | MapReduce | PySpark |
| --- | --- | --- |
| Ease of Use | Harder, lower-level API | Easier, higher-level Python API |
| Performance | Slower (b/c disk-based) | Faster (b/c in-memory processing) |
| Fault Tolerance | Achieve using task re-execution | Achieve by using RDD-based fault  Tolerance (RDD have lineage) |
| Processing Model | Batch processing only | Batch + Real-time stream processing |
| Scalability | Good, but I/O-heavy | Excellent, more efficient |
| Ecosystem | Hadoop ecosystem (HDFS, Hive, etc.) | Spark ecosystem, integrated tools |
| Best For | Simple, large-scale batch processing | Complex, iterative, real-time, ML |

**###############################################################################**

**# PYSPARK and PANDAS #**

**###############################################################################**

We can discuss the difference between pyspark and pandas on below points-

1. Scalability
2. Performance on Large Datasets
3. Distributed Computing
4. Fault Tolerance
5. Integration with Big Data Ecosystem
6. Memory Management
7. Data Parallelism
8. SQL-like Operations
9. DataFrame API

**1. Scalability**

* **PySpark**: It is designed to handle large datasets that exceed the memory of a single machine by leveraging the power of distributed computing. It can run on clusters of machines, so you can scale up your processing power as needed.
* **Pandas**: Works on a single machine and loads the entire dataset into memory. This makes it unsuitable for very large datasets that can't fit into RAM.

**2. Performance on Large Datasets**

* **PySpark**: Processes data in parallel across multiple nodes in a cluster, making it significantly faster for large datasets (terabytes or more). PySpark also allows for fault tolerance and data locality, making it more efficient for distributed systems.
* **Pandas**: Performance can degrade significantly as the dataset size increases because it relies on a single machine and memory to process data.

**3. Distributed Computing**

* **PySpark**: Built on Apache Spark, it enables distributed computing by breaking up tasks and data across many machines in a cluster. This allows PySpark to handle distributed data processing and operations such as joins, aggregations, and more at scale.
* **Pandas**: Does not natively support distributed computing. While you can use libraries like Dask to parallelize tasks in pandas, this requires extra setup and doesn't scale as efficiently as PySpark.

**4. Fault Tolerance**

* **PySpark**: Since it's built on Spark, PySpark has built-in fault tolerance. If a node in a cluster fails, the data and computations are automatically recovered, ensuring that processing continues smoothly.
* **Pandas**: Does not have fault tolerance. If an error occurs or a machine fails, the process will stop.

**5. Integration with Big Data Ecosystem**

* **PySpark**: Works seamlessly with Hadoop and other big data tools, allowing for the handling of big data workloads. It supports reading from and writing to various data formats like HDFS, S3, Parquet, and more.
* **Pandas**: Primarily works with data loaded from local files or databases. It can handle CSV, Excel, or SQL-based data sources, but it doesn't have direct support for Hadoop or other distributed systems.

**6. Memory Management**

* **PySpark**: Operates with lazy evaluation, meaning computations are not executed until an action is triggered (e.g., collect, save). This allows Spark to optimize resource usage and manage memory more efficiently in large-scale data processing.
* **Pandas**: Operates with eager evaluation, meaning computations are executed immediately, and large datasets can quickly consume all available memory.

**7. Data Parallelism**

* **PySpark**: Supports distributed data processing across multiple machines, meaning operations like filtering, aggregating, and joining data are done in parallel, which increases throughput and reduces processing time for large datasets.
* **Pandas**: Operates sequentially on data in memory, so it does not benefit from the same level of parallelism and distributed computation.

**8. SQL-like Operations**

* **PySpark**: Provides an SQL interface for performing data transformations using Spark SQL, which can be more intuitive for people familiar with SQL.
* **Pandas**: Provides data manipulation through DataFrame methods but does not have an SQL-like query interface built-in.

**9. DataFrame API**

* **PySpark**: The PySpark DataFrame API is built for distributed computing, which means it's optimized for performance in a big data environment.
* **Pandas**: The pandas DataFrame is built for in-memory, single-machine data manipulation, and while it offers a rich set of operations, it's limited in terms of scalability.

**###############################**

**# Characteristic of RDD #**

**###############################**

* It is a distributed collection of objects, i.e. each RDD is divided into multiple partitions and each of these partitions can reside in memory or stored on the disk of different machines in a cluster.
* They are immutable (read-only) and so they cannot be modified. A new RDD will be created every time some transformation is done on an existing RDD.
* RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

**What is partition**

If datasets are huge then cannot fit on single node and they have to be partitioned across different node or machine.

Partition in spark is basically as atomic chunk of data stored on a node in cluster. They are basic units of parallelism.

One partition cann’t span over multiple machine.

**Different way of creating RDD**

We can create RDD from below different way-

1. From Externa Data ( HDFS, local etc )
2. Local Data
3. Python List/ Parallelized collection
4. Other RDD
5. Existing DataFrame

**##########################**

**# Hands-On Exercise on RDD #**

**##########################**

**Split operation based on given string of RDD row.**

We can perform split operation of RDD row using map with a function (function\_name) for splitting.

rdd\_data.map ( function\_name )

**Exercise 1**:

Read the iris\_site.csv file and convert into RDD and get count rows in it.

**Solution 1:**

iris1=sc.textFile('D:\\PySpark\\Prac1\\iris\\iris\_site.csv')

print(iris1.count())

**Solution 2:** --- Read as dataframe and then convert into rdd

df1=spark.read.csv("D:\\PySpark\\Prac1\\iris\\iris\_site.csv")

print(type(df1.rdd))

**Exercise 2:**

When we create the RDD, each row is coming as one string, split the row data against comma and create column.

**Solution:**

iris1=sc.textFile('D:\\PySpark\\Prac1\\iris\\iris\_site.csv')

iris\_split = iris1.map( lambda x : x.split( ',' )

**############################**

**# Extracting given column data #**

**############################**

We can extract the given column index data using map –

extracted\_rdd = rdd\_data.map( lambda x : x[ n ] ) --- n= index of column which want to extract.

**Exercise 3:**

Extract the 2nd column or column at index 1 from iris\_rdd data which is splitted on comma.

**Solution:**

iris1\_mod = iris1\_split.map(lambda x:x[0])

print(iris1\_mod.take(10))

**##################################**

**# Changing datatype of a column in RDD #**

**##################################**

**Exercise:**

Change the second column of above rdd to float type.

Solution:

iris\_split=iris\_split.map( lambda x :float(x[1]) )

**Exercise:**

Change the data type of all column to float of RDD

**Solution:**

iris\_split=iris\_split.map( lambda x : float(x[0]), float(x[1]) . . . . float(x[n] ) )

**######################**

**# Creating Key-Value Pair #**

**######################**

We can create key-value pair by using map function and flatmap function

**Example:**

Suppose we can to create a key-value pair for first 4 column of iris\_split rdd.

Keys= – key1, key2, key3, ley4

Value of above key = col1, col2, col3, col4 data

**Solution: --- Using map()**

Solution:

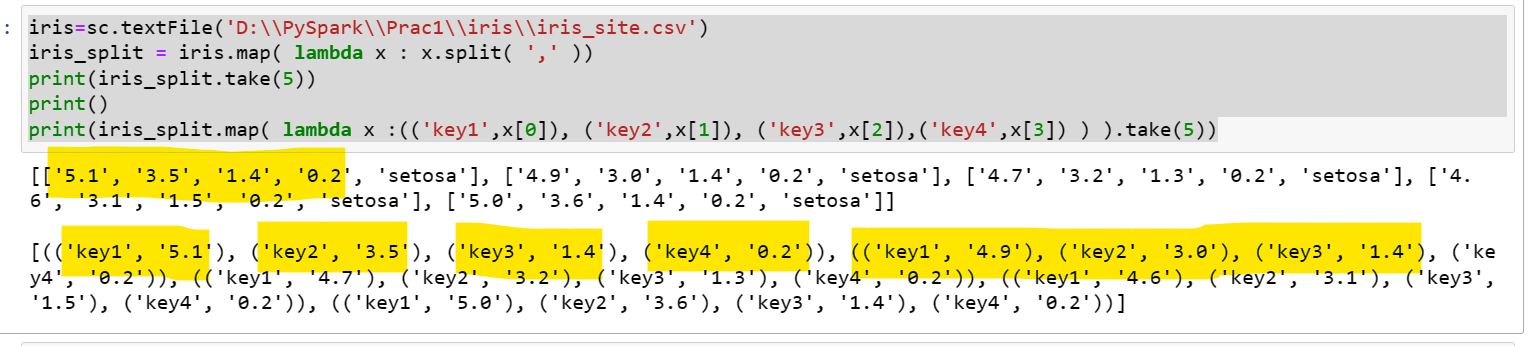
iris=sc.textFile('D:\\PySpark\\Prac1\\iris\\iris\_site.csv')

iris\_split = iris.map( lambda x : x.split( ',' ))

print(iris\_split.take(5))

print()

print(iris\_split.map( lambda x :(('key1',x[0]), ('key2',x[1]), ('key3',x[2]),('key4',x[3]) ) ).take(5))



**Note:**

It can be observed that the key-value pairs have been created. However, it can also be noted that the entire key-value collection is present inside a double collection.

**Solution: Using flatmap()**

iris=sc.textFile('D:\\PySpark\\Prac1\\iris\\iris\_site.csv')

iris\_split = iris.map( lambda x : x.split( ',' ))

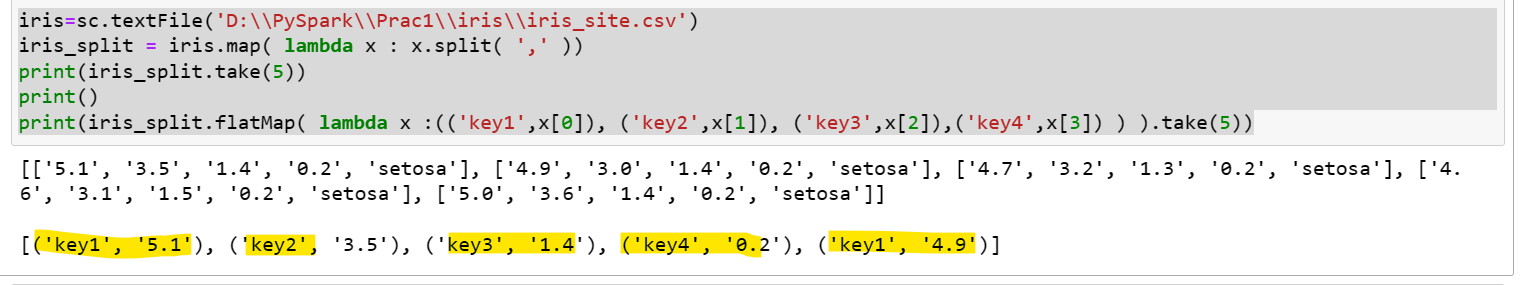
print(iris\_split.take(5))

print()

print(iris\_split.flatMap( lambda x :(('key1',x[0]), ('key2',x[1]), ('key3',x[2]),('key4',x[3]) ) ).take(5))

Note:

Here key-value pair will not be collection of collection, if will be like create collection then change in 1-d list. flatten operation of numpy



##################

# map() vs flatmap() #

##################

<https://medium.com/@sujathamudadla1213/what-is-the-difference-between-map-and-flatmap-in-spark-6e6fa9360094>

|  |  |
| --- | --- |
| flatmap | map |
| In flatmap applies on each element of input RDD/Dataframe and returns new RDD/Dataframe where each input element can be mapped to zero or more output elements | In flatmap applies on each element of input RDD/Dataframe and returns RDD/Dataframe where each element corresponds to the result of applying the function to the corresponding element of the original RDD or DataFrame |
| flatMap can return output a sequence or iterable of values for each input element, and these values will be concatenated into the output RDD or DataFrame i.e output will be one iterable data | The output of the map transformation has a one-to-one mapping between input and output elements. In other words, if you start with an RDD or DataFrame of n elements, the map transformation will produce a new RDD or DataFrame with the **same number of elements** ( each input will have one output data which can be iterable also) |

################

# Sorting of RDD #

################

We can sort the data of RDD using **sortBy()** function for simple sort and **soryByKey()** for RDD of key-value type.

sortBy --- for simple sort

soryByKey --- for kay-value pair sorting.

**Example:**

Sort the data of iris\_split rdd on first column value.

**Solution : - normal sorting**

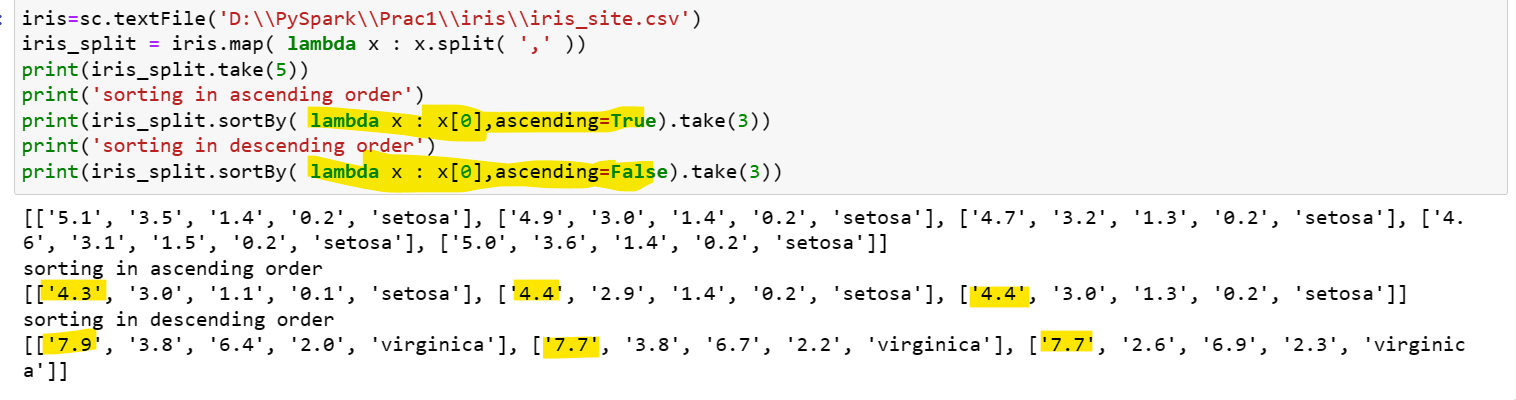
iris=sc.textFile('D:\\PySpark\\Prac1\\iris\\iris\_site.csv')

iris\_split = iris.map( lambda x : x.split( ',' ))

print(iris\_split.take(5))

print()

print(iris\_split.sortBy( lambda x : x[0]).take(3))



**Solutuon 2: Sorting by key**

iris=sc.textFile('D:\\PySpark\\Prac1\\iris\\iris\_site.csv')

iris1\_split = iris.map(lambda var1: var1.split(","))

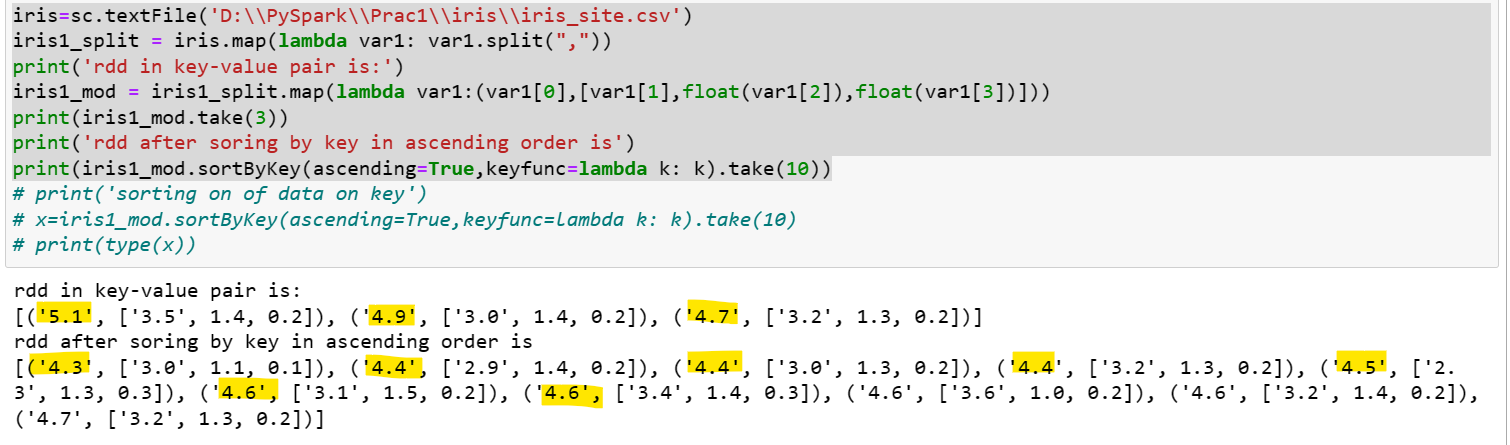
print('rdd in key-value pair is:')

iris1\_mod = iris1\_split.map(lambda var1:(var1[0],[var1[1],float(var1[2]),float(var1[3])]))

print(iris1\_mod.take(3))

print('rdd after soring by key in ascending order is')

print( iris1\_mod.sortByKey(ascending=True,keyfunc=lambda k: k).take(10) )



**##########################**

**# Combining/Union of RDD #**

**##########################**

For combining or performing union of two or more RDD can be done by using union function

rdd1.union(rdd2, rdd3, . . . rddn)

**Example:**

iris=sc.textFile('D:\\PySpark\\Prac1\\iris\\iris\_site.csv')

iris1\_split = iris.map(lambda var1: var1.split(","))

#Creating RDDs

rdd1 = iris1\_split.map(lambda var1: ['Sepal.Length',float(var1[0])])

rdd2 = iris1\_split.map(lambda var1: ['Sepal.Width',float(var1[1])])

rdd3 = iris1\_split.map(lambda var1: ['Petal.Length',float(var1[2])])

print('now rdds are:')

print(SL.take(2))

print(SW.take(2))

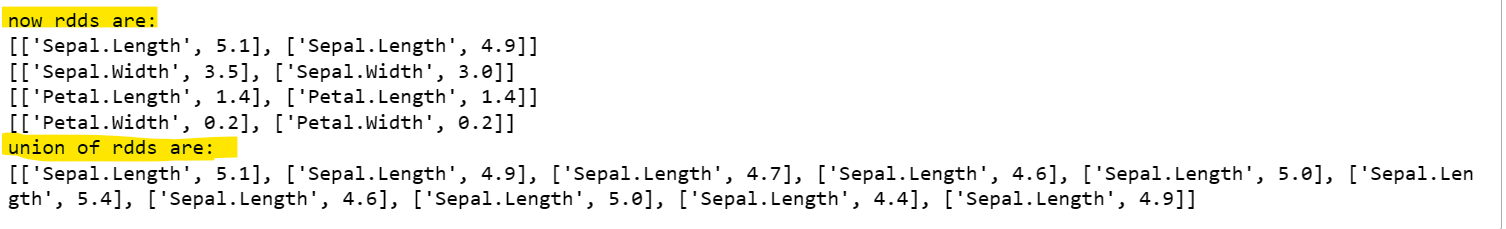
print(PL.take(2))

print(PW.take(2))

union\_data = sc.union([SL,SW,PL,PW])

print('union of rdds are:')

print(union\_data.take(10))



**###################**

**# Intersection of RDD #**

**###################**

For getting intersection of two or more RDD data we can get it by using intersection() function of rdd.

rdd1.intersection(rdd2)

**Example:**

EmployeeList = sc.parallelize(['a','b','c','d','e']) #creating first rdd

HealthMemberShip = sc.parallelize(['d','e','f','g']) # creating second rdd

print(EmployeeList.intersection(HealthMemberShip)) #create intersection of rdd

Example2:

Find the customer count who order in July and august month.

Solution:

sc=spark.sparkContext

#D:\Udemy\_PySpark\RetailDB+SalesData\RetailDB SalesData\Orders\part-00000

rdd=sc.textFile('D:\\Udemy\_PySpark\\RetailDB+SalesData\\RetailDBSalesData\\Orders\\order.txt')

#ger the customer for july month

rddJuly=rdd.filter(lambda x : (x.split(',')[1].split('-')[1]=='07')).map(lambda x : x.split(',')[2])

print('number of costomer for July month: ',rddJuly.count())

#ger the customer for aug month

rddAug=rdd.filter(lambda x : (x.split(',')[1].split('-')[1]=='08')).map(lambda x : x.split(',')[2])

#ger the common  customer for july and aug month

print('number of costomer for Aug month: ',rddAug.count())

rddCommon=rddJuly.intersection(rddAug)

print('common customer count for aug and july month: ',rddCommon.count())

**#######################**

**# Creating Join of RDD #**

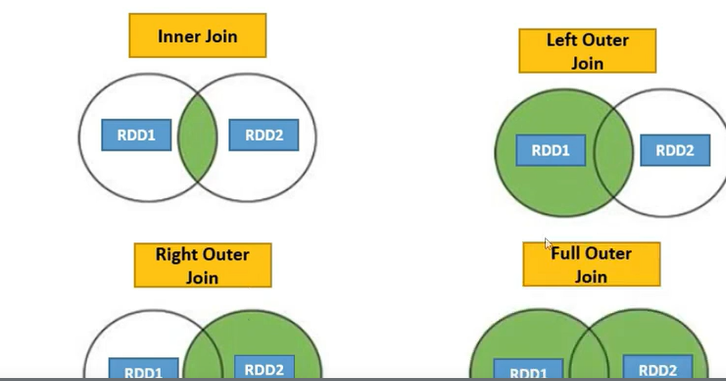
**#######################**

We can perform join operation of two RDD using join function of RDD.

Joining of RDD is possible when RDD is of KEY-VALUE type.

In pyspark RDD, join are of below type-

* Join or inner join
* Leftouter join
* RightOuter join
* fullOuter join



**#Inner join #**

rdd1.join(rdd2, numPartitions= None)

**Note:**

* When join method is called on type (K, V) and (K, W) types then it return the result of (K, (V, W)) pairs with all pairs of elements for each key.
* For doing join operation we should convert the RDD into key-value pair first.

**Example:**

#creating RDD1

location1 = sc.parallelize([('employee1',[3,1,2,5,4]),('employee2',[2,4,1,2,2])])

#creating RDD2

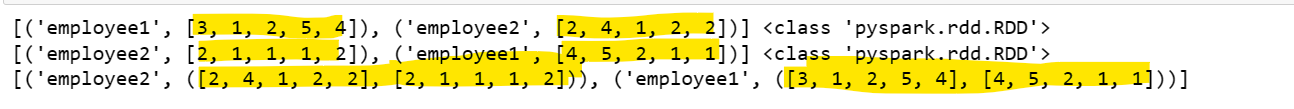
location2 = sc.parallelize([('employee2',[2,1,1,1,2]),('employee1',[4,5,2,1,1])])

#Performing join operation

**join1 = location1.join(location2)**

print(join1.take(10))

**Result:**



**rdd1.leftOuterJoin(rdd2, numPartitions= None)** – for left outer join

**rdd1.rightOuterJoin(rdd2, numPartitions= None)** – for right outer join

**rdd1.fullOuterJoin(rdd2, numPartitions= None)** – for full outer join

**Note:**

When join method is called on type (K, V) and (K, W) types then it retunr the result of (K, (V, W)) pairs with all pairs of elements for each key

**#####################################**

**# Applying a function to each value of RDD #**

**#####################################**

In case if we can to apply a given function to each value of RDD then we can use 'foreach' function. ----- **Don’t use for any transformation purpose better go for map().**

rdd.foreach( my\_func\_name )

################

# Filtering of RDD #

################

For filtering purpose, we can use filter function of rdd

rdd.filter( filter\_condition )

**Example:**

#create rdd from file

iris=sc.textFile('D:\\PySpark\\Prac1\\iris\\iris\_site.csv')

#splitting each row of rdd against comma

iris1\_split = iris1.map(lambda var1: var1.split(","))

#getting only those rdd for which first col is gt 7

filter1 = iris1\_split.filter(lambda x: float(x[0]) > 7)

################

# Methods of RDD #

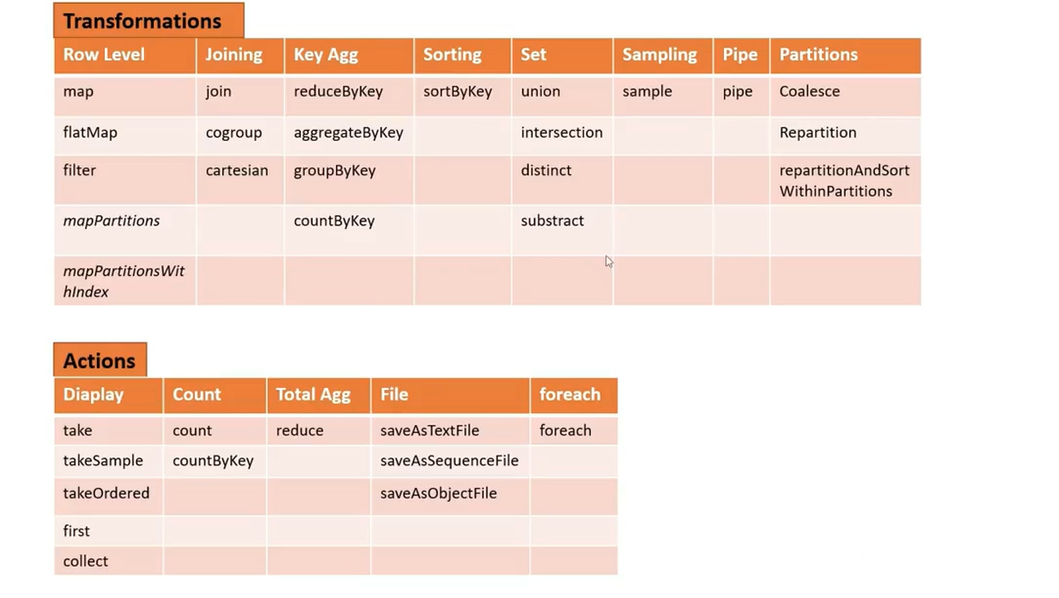
################

We have below RDD function available.

|  |  |
| --- | --- |
| [RDD.aggregate](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.aggregate.html#pyspark.RDD.aggregate)(zeroValue, seqOp, combOp) | Aggregate the elements of each partition, and then the results for all the partitions, using a given combine functions and a neutral “zero value.” |
| [RDD.aggregateByKey](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.aggregateByKey.html#pyspark.RDD.aggregateByKey)(zeroValue, seqFunc, combFunc) | Aggregate the values of each key, using given combine functions and a neutral “zero value”. |
| [RDD.barrier](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.barrier.html#pyspark.RDD.barrier)() | Marks the current stage as a barrier stage, where Spark must launch all tasks together. |
| [RDD.cache](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.cache.html#pyspark.RDD.cache)() | Persist this RDD with the default storage level (*MEMORY\_ONLY*). |
| [RDD.cartesian](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.cartesian.html#pyspark.RDD.cartesian)(other) | Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements (a, b) where a is in *self* and b is in *other*. |
| [RDD.checkpoint](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.checkpoint.html#pyspark.RDD.checkpoint)() | Mark this RDD for checkpointing. |
| [RDD.coalesce](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.coalesce.html#pyspark.RDD.coalesce)(numPartitions[, shuffle]) | Return a new RDD that is reduced into *numPartitions* partitions. |
| [RDD.cogroup](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.cogroup.html#pyspark.RDD.cogroup)(other[, numPartitions]) | For each key k in *self* or *other*, return a resulting RDD that contains a tuple with the list of values for that key in *self* as well as *other*. |
| [RDD.collect](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.collect.html#pyspark.RDD.collect)()  Returns rows of RDD in list form  In df collect() function returns the row of df in list form | Return a list that contains all of the elements in this RDD. Rdd rows in list form |
| [RDD.collectAsMap](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.collectAsMap.html#pyspark.RDD.collectAsMap)() | Return the key-value pairs in this RDD to the master as a dictionary. |
| [RDD.collectWithJobGroup](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.collectWithJobGroup.html#pyspark.RDD.collectWithJobGroup)(groupId, description) | When collect rdd, use this method to specify job group. |
| [RDD.combineByKey](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.combineByKey.html#pyspark.RDD.combineByKey)(createCombiner, mergeValue, …) | Generic function to combine the elements for each key using a custom set of aggregation functions. |
| [RDD.context](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.context.html#pyspark.RDD.context) | The [SparkContext](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.SparkContext.html#pyspark.SparkContext) that this RDD was created on. |
| [RDD.count](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.count.html#pyspark.RDD.count)() | Return the number of elements in this RDD. |
| [RDD.countApprox](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.countApprox.html#pyspark.RDD.countApprox)(timeout[, confidence]) | Approximate version of count() that returns a potentially incomplete result within a timeout, even if not all tasks have finished. |
| [RDD.countApproxDistinct](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.countApproxDistinct.html#pyspark.RDD.countApproxDistinct)([relativeSD]) | Return approximate number of distinct elements in the RDD. |
| [RDD.countByKey](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.countByKey.html#pyspark.RDD.countByKey)() | Count the number of elements for each key, and return the result to the master as a dictionary. |
| [RDD.countByValue](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.countByValue.html#pyspark.RDD.countByValue)() | Return the count of each unique value in this RDD as a dictionary of (value, count) pairs. |
| [RDD.distinct](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.distinct.html#pyspark.RDD.distinct)([numPartitions]) | Return a new RDD containing the distinct elements in this RDD. |
| [RDD.filter](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.filter.html#pyspark.RDD.filter)(f)  f=function used for filtration | Return a new RDD containing only the elements that satisfy a predicate. |
| [RDD.first](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.first.html#pyspark.RDD.first)()  This function is in dataframe also.  Return the first element of RDD/DF | Return the first element in this RDD. |
| [RDD.flatMap](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.flatMap.html#pyspark.RDD.flatMap)(f[, preservesPartitioning]) | Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results. |
| [RDD.flatMapValues](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.flatMapValues.html#pyspark.RDD.flatMapValues)(f) | Pass each value in the key-value pair RDD through a flatMap function without changing the keys; this also retains the original RDD’s partitioning. |
| [RDD.fold](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.fold.html#pyspark.RDD.fold)(zeroValue, op) | Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral “zero value.” |
| [RDD.foldByKey](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.foldByKey.html#pyspark.RDD.foldByKey)(zeroValue, func[, …]) | Merge the values for each key using an associative function “func” and a neutral “zeroValue” which may be added to the result an arbitrary number of times, and must not change the result (e.g., 0 for addition, or 1 for multiplication.). |
| [RDD.foreach](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.foreach.html#pyspark.RDD.foreach)(f) | Applies a function to all elements of this RDD. |
| [RDD.foreachPartition](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.foreachPartition.html#pyspark.RDD.foreachPartition)(f) | Applies a function to each partition of this RDD. |
| [RDD.fullOuterJoin](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.fullOuterJoin.html#pyspark.RDD.fullOuterJoin)(other[, numPartitions]) | Perform a right outer join of *self* and *other*. |
| [RDD.getCheckpointFile](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.getCheckpointFile.html#pyspark.RDD.getCheckpointFile)() | Gets the name of the file to which this RDD was checkpointed |
| [RDD.getNumPartitions](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.getNumPartitions.html#pyspark.RDD.getNumPartitions)() | Returns the number of partitions in RDD |
| [RDD.getResourceProfile](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.getResourceProfile.html#pyspark.RDD.getResourceProfile)() | Get the [pyspark.resource.ResourceProfile](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.resource.ResourceProfile.html#pyspark.resource.ResourceProfile) specified with this RDD or None if it wasn’t specified. |
| [RDD.getStorageLevel](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.getStorageLevel.html#pyspark.RDD.getStorageLevel)() | Get the RDD’s current storage level. |
| [RDD.glom](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.glom.html#pyspark.RDD.glom)() | Return an RDD created by coalescing all elements within each partition into a list. |
| [RDD.groupBy](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.groupBy.html#pyspark.RDD.groupBy)(f[, numPartitions, partitionFunc]) | Return an RDD of grouped items. |
| [RDD.groupByKey](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.groupByKey.html#pyspark.RDD.groupByKey)([numPartitions, partitionFunc]) | Group the values for each key in the RDD into a single sequence. |
| [RDD.groupWith](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.groupWith.html#pyspark.RDD.groupWith)(other, \*others) | Alias for cogroup but with support for multiple RDDs. |
| [RDD.histogram](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.histogram.html#pyspark.RDD.histogram)(buckets) | Compute a histogram using the provided buckets. |
| [RDD.id](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.id.html#pyspark.RDD.id)() | A unique ID for this RDD (within its SparkContext). |
| [RDD.intersection](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.intersection.html#pyspark.RDD.intersection)(other) | Return the intersection of this RDD and another one. |
| [RDD.isCheckpointed](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.isCheckpointed.html#pyspark.RDD.isCheckpointed)() | Return whether this RDD is checkpointed and materialized, either reliably or locally. |
| [RDD.isEmpty](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.isEmpty.html#pyspark.RDD.isEmpty)() | Returns true if and only if the RDD contains no elements at all. |
| [RDD.isLocallyCheckpointed](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.isLocallyCheckpointed.html#pyspark.RDD.isLocallyCheckpointed)() | Return whether this RDD is marked for local checkpointing. |
| [RDD.join](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.join.html#pyspark.RDD.join)(other[, numPartitions]) | Return an RDD containing all pairs of elements with matching keys in *self* and *other*. |
| [RDD.keyBy](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.keyBy.html#pyspark.RDD.keyBy)(f) | Creates tuples of the elements in this RDD by applying *f*. |
| [RDD.keys](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.keys.html#pyspark.RDD.keys)() | Return an RDD with the keys of each tuple. |
| [RDD.leftOuterJoin](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.leftOuterJoin.html#pyspark.RDD.leftOuterJoin)(other[, numPartitions]) | Perform a left outer join of *self* and *other*. |
| [RDD.localCheckpoint](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.localCheckpoint.html#pyspark.RDD.localCheckpoint)() | Mark this RDD for local checkpointing using Spark’s existing caching layer. |
| [RDD.lookup](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.lookup.html#pyspark.RDD.lookup)(key) | Return the list of values in the RDD for key *key*. |
| [RDD.map](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.map.html#pyspark.RDD.map)(f[, preservesPartitioning]) | Return a new RDD by applying a function to each element of this RDD. |
| [RDD.mapPartitions](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.mapPartitions.html#pyspark.RDD.mapPartitions)(f[, preservesPartitioning]) | Return a new RDD by applying a function to each partition of this RDD. |
| [RDD.mapPartitionsWithIndex](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.mapPartitionsWithIndex.html#pyspark.RDD.mapPartitionsWithIndex)(f[, …]) | Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition. |
| [RDD.mapPartitionsWithSplit](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.mapPartitionsWithSplit.html#pyspark.RDD.mapPartitionsWithSplit)(f[, …]) | Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition. |
| [RDD.mapValues](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.mapValues.html#pyspark.RDD.mapValues)(f)  Applies the given function for each key in key-value pair rdd and return – [(key, result\_of\_applied\_func) ] | Pass each value in the key-value pair RDD through a map function without changing the keys; this also retains the original RDD’s partitioning. |
| [RDD.max](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.max.html#pyspark.RDD.max)([key]) | Find the maximum item in this RDD. |
| [RDD.mean](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.mean.html#pyspark.RDD.mean)() | Compute the mean of this RDD’s elements. |
| [RDD.meanApprox](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.meanApprox.html#pyspark.RDD.meanApprox)(timeout[, confidence]) | Approximate operation to return the mean within a timeout or meet the confidence. |
| [RDD.min](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.min.html#pyspark.RDD.min)([key]) | Find the minimum item in this RDD. |
| [RDD.name](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.name.html#pyspark.RDD.name)() | Return the name of this RDD. |
| [RDD.partitionBy](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.partitionBy.html#pyspark.RDD.partitionBy)(numPartitions[, partitionFunc]) | Return a copy of the RDD partitioned using the specified partitioner. |
| [RDD.persist](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.persist.html#pyspark.RDD.persist)([storageLevel]) | Set this RDD’s storage level to persist its values across operations after the first time it is computed. |
| [RDD.pipe](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.pipe.html#pyspark.RDD.pipe)(command[, env, checkCode]) | Return an RDD created by piping elements to a forked external process. |
| [RDD.randomSplit](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.randomSplit.html#pyspark.RDD.randomSplit)(weights[, seed]) | Randomly splits this RDD with the provided weights. |
| [RDD.reduce](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.reduce.html#pyspark.RDD.reduce)(f) | Reduces the elements of this RDD using the specified commutative and associative binary operator. |
| [RDD.reduceByKey](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.reduceByKey.html#pyspark.RDD.reduceByKey)(func[, numPartitions, …]) | Merge the values for each key using an associative and commutative reduce function. |
| [RDD.reduceByKeyLocally](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.reduceByKeyLocally.html#pyspark.RDD.reduceByKeyLocally)(func) | Merge the values for each key using an associative and commutative reduce function, but return the results immediately to the master as a dictionary. |
| [RDD.repartition](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.repartition.html#pyspark.RDD.repartition)(numPartitions) | Return a new RDD that has exactly numPartitions partitions. |
| [RDD.repartitionAndSortWithinPartitions](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.repartitionAndSortWithinPartitions.html#pyspark.RDD.repartitionAndSortWithinPartitions)([…]) | Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. |
| [RDD.rightOuterJoin](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.rightOuterJoin.html#pyspark.RDD.rightOuterJoin)(other[, numPartitions]) | Perform a right outer join of *self* and *other*. |
| [RDD.sample](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.sample.html#pyspark.RDD.sample)(withReplacement, fraction[, seed]) | Return a sampled subset of this RDD. |
| [RDD.sampleByKey](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.sampleByKey.html#pyspark.RDD.sampleByKey)(withReplacement, fractions) | Return a subset of this RDD sampled by key (via stratified sampling). |
| [RDD.sampleStdev](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.sampleStdev.html#pyspark.RDD.sampleStdev)() | Compute the sample standard deviation of this RDD’s elements (which corrects for bias in estimating the standard deviation by dividing by N-1 instead of N). |
| [RDD.sampleVariance](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.sampleVariance.html#pyspark.RDD.sampleVariance)() | Compute the sample variance of this RDD’s elements (which corrects for bias in estimating the variance by dividing by N-1 instead of N). |
| [RDD.saveAsHadoopDataset](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.saveAsHadoopDataset.html#pyspark.RDD.saveAsHadoopDataset)(conf[, …]) | Output a Python RDD of key-value pairs (of form RDD[(K, V)]) to any Hadoop file system, using the old Hadoop OutputFormat API (mapred package). |
| [RDD.saveAsHadoopFile](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.saveAsHadoopFile.html#pyspark.RDD.saveAsHadoopFile)(path, outputFormatClass) | Output a Python RDD of key-value pairs (of form RDD[(K, V)]) to any Hadoop file system, using the old Hadoop OutputFormat API (mapred package). |
| [RDD.saveAsNewAPIHadoopDataset](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.saveAsNewAPIHadoopDataset.html#pyspark.RDD.saveAsNewAPIHadoopDataset)(conf[, …]) | Output a Python RDD of key-value pairs (of form RDD[(K, V)]) to any Hadoop file system, using the new Hadoop OutputFormat API (mapreduce package). |
| [RDD.saveAsNewAPIHadoopFile](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.saveAsNewAPIHadoopFile.html#pyspark.RDD.saveAsNewAPIHadoopFile)(path, …[, …]) | Output a Python RDD of key-value pairs (of form RDD[(K, V)]) to any Hadoop file system, using the new Hadoop OutputFormat API (mapreduce package). |
| [RDD.saveAsPickleFile](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.saveAsPickleFile.html#pyspark.RDD.saveAsPickleFile)(path[, batchSize]) | Save this RDD as a SequenceFile of serialized objects. |
| [RDD.saveAsSequenceFile](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.saveAsSequenceFile.html#pyspark.RDD.saveAsSequenceFile)(path[, …]) | Output a Python RDD of key-value pairs (of form RDD[(K, V)]) to any Hadoop file system, using the “org.apache.hadoop.io.Writable” types that we convert from the RDD’s key and value types. |
| [RDD.saveAsTextFile](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.saveAsTextFile.html#pyspark.RDD.saveAsTextFile)(path[, compressionCodecClass]) | Save this RDD as a text file, using string representations of elements. |
| [RDD.setName](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.setName.html#pyspark.RDD.setName)(name) | Assign a name to this RDD. |
| [RDD.sortBy](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.sortBy.html#pyspark.RDD.sortBy)(keyfunc[, ascending, numPartitions]) | Sorts this RDD by the given keyfunc |
| [RDD.sortByKey](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.sortByKey.html#pyspark.RDD.sortByKey)([ascending, numPartitions, …]) | Sorts this RDD, which is assumed to consist of (key, value) pairs. |
| [RDD.stats](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.stats.html#pyspark.RDD.stats)() | Return a StatCounter object that captures the mean, variance and count of the RDD’s elements in one operation. |
| [RDD.stdev](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.stdev.html#pyspark.RDD.stdev)() | Compute the standard deviation of this RDD’s elements. |
| [RDD.subtract](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.subtract.html#pyspark.RDD.subtract)(other[, numPartitions])  Get the result after removing the rdd2 data  from rdd1 data | Return each value in *self* that is not contained in *other*. |
| [RDD.subtractByKey](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.subtractByKey.html#pyspark.RDD.subtractByKey)(other[, numPartitions]) | Return each (key, value) pair in *self* that has no pair with matching key in *other*. |
| [RDD.sum](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.sum.html#pyspark.RDD.sum)() | Add up the elements in this RDD. |
| [RDD.sumApprox](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.sumApprox.html#pyspark.RDD.sumApprox)(timeout[, confidence]) | Approximate operation to return the sum within a timeout or meet the confidence. |
| [RDD.take](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.take.html#pyspark.RDD.take)(num) | Take the first num elements of the RDD. |
| [RDD.takeOrdered](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.takeOrdered.html#pyspark.RDD.takeOrdered)(num[, key])  Num= number of elements to fetch  Key = function for ordering | Get the N elements from an RDD ordered in ascending order or as specified by the optional key function. |
| [RDD.takeSample](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.takeSample.html#pyspark.RDD.takeSample)(withReplacement, num[, seed]) | Return a fixed-size sampled subset of this RDD. |
| [RDD.toDebugString](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.toDebugString.html#pyspark.RDD.toDebugString)() | A description of this RDD and its recursive dependencies for debugging. |
| [RDD.toLocalIterator](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.toLocalIterator.html#pyspark.RDD.toLocalIterator)([prefetchPartitions]) | Return an iterator that contains all of the elements in this RDD. |
| [RDD.top](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.top.html#pyspark.RDD.top)(num[, key]) | Get the top N elements from an RDD. |
| [RDD.treeAggregate](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.treeAggregate.html#pyspark.RDD.treeAggregate)(zeroValue, seqOp, combOp) | Aggregates the elements of this RDD in a multi-level tree pattern. |
| [RDD.treeReduce](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.treeReduce.html#pyspark.RDD.treeReduce)(f[, depth]) | Reduces the elements of this RDD in a multi-level tree pattern. |
| [RDD.union](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.union.html#pyspark.RDD.union)(other) | Return the union of this RDD and another one. |
| [RDD.unpersist](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.unpersist.html#pyspark.RDD.unpersist)([blocking]) | Mark the RDD as non-persistent, and remove all blocks for it from memory and disk. |
| [RDD.values](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.values.html#pyspark.RDD.values)() | Return an RDD with the values of each tuple. |
| [RDD.variance](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.variance.html#pyspark.RDD.variance)() | Compute the variance of this RDD’s elements. |
| [RDD.withResources](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.withResources.html#pyspark.RDD.withResources)(profile) | Specify a [pyspark.resource.ResourceProfile](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.resource.ResourceProfile.html#pyspark.resource.ResourceProfile) to use when calculating this RDD. |
| [RDD.zip](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.zip.html#pyspark.RDD.zip)(other) | Zips this RDD with another one, returning key-value pairs with the first element in each RDD second element in each RDD, etc. |
| [RDD.zipWithIndex](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.zipWithIndex.html#pyspark.RDD.zipWithIndex)() | Zips this RDD with its element indices. |
| [RDD.zipWithUniqueId](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.RDD.zipWithUniqueId.html#pyspark.RDD.zipWithUniqueId)() | Zips this RDD with generated unique Long ids |

In RDD we can categorize the RDD function in two type-

1. Transformataions
2. Actions



Commonly used function description

**rdd2=rdd1.filter( filtering\_condition )**

Return the filtered rdd

We can apply multiple filters as below-

* rdd.filter( condition1 | condition2 | ... condition N )
* rdd.filter( condition1 & condition2 & ... condition N )
* rdd.filter( condition1 & condition2 | ... condition N )

**rdd2=.distinct(numPartitions=None)**

Retunt the unique values from rdd as new rdd

Example:

sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())

Question:

Find the distinct value in 5th column of isiri rdd.

Solution:

iris1\_split = iris1.map(lambda var1: var1.split(","))

iris1\_mod = iris1\_split.map(lambda col:col[4])

**Rdd.mapValues(func)**

Only applicable for RDD which is in key-value form.

Passes each value in the key-value pair RDD through a given map function(func) without changing the keys.

It return the data as (key,mapped\_value from function(func) )as rdd

this also retains the original RDD’s partitioning.

It applies the given function on key-value type rdd, making no change in key.

|  |  |
| --- | --- |
| map() | mapvalues( func ) |
| * applies given given function on all values. * When RDD is in normal RDD | * Applied given function on values of given key * When RDD is in form of key-value pair. |

**Question:**

Find the number of data for each key of a rdd.

**Solution:**

x = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b", ["grapes"])])

def f(x):

return len(x)

x.mapValues(f).collect()

[('a', 3), ('b', 1)]

**rdd.count()**

Return the number of element/rows in rdd

**rdd.countByKey()**

Count the number of elements for each key, and return the result to the master as a **dictionary**

Example:

rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

sorted(rdd.countByKey().items()) #[('a', 2), ('b', 1)]

**rdd.countByValue()**

Return the count of each unique value in this RDD as a dictionary of (value, count) pairs.

Example 1:

sorted(sc.parallelize([1, 2, 1, 2, 2], 2).countByValue().items()) #[(1, 2), (2, 3)]

**Question:**

Count the occurrence of each value in fith column of iris rdd

iris1\_split = iris1.map(lambda var1: var1.split(","))

print(iris1\_split.map(lambda col: col[4]).countByValue().items())

**rdd.isEmpty()**

Returns True if RDD is empty

**rdd1.join(other, numPartitions=None)**

Return an RDD containing all pairs of elements with matching keys in self and other.

This is basicaly used for joing two rdd if data are in form of key-valye pair.

Joins will happen on common key column name

x = sc.parallelize([("a", 1), ("b", 4)])

y = sc.parallelize([("a", 2), ("a", 3)])

#both rdd have common col name 'a' so join will happend on key 'a'

sorted(x.join(y).collect())

[('a', (1, 2)), ('a', (1, 3))]

**rdd.sortBy(keyfunc, ascending=True, numPartitions=None)**

Sorts this RDD by the given keyfunc.

Example:

tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]

sc.parallelize(tmp).sortBy(lambda x: x[0]).collect()

**rdd1.groupWith(rdd2, rdd3, . . , rddn)**

**rdd.groupBy(f: Callable[[T], K], numPartitions: Optional[int] = None)**

Return an RDD of grouped items.

**Question**:

Suppose we have a rdd and want to group that rdd on first three value of column data.

**Solution**

rdd4 = rdd3.groupBy(lambda w: w[0:3])

**rdd.groupByKey(numPartitions: Optional[int] = None, partitionFunc: Callable[[K])**

Group the values for each key in the RDD into a single sequence. Hash-partitions the resulting RDD with numPartitions partitions.

Returns list of tuple –

[ key1, [val1, val2 … valn], key2, [ valu1, valu2, . . valun]

Key<n> – key from RDD key-value pair

[ val1, val2, . . valn ] – all value as list for key in rdd element

**Example**:

rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])

sorted(rdd.groupByKey().mapValues(len).collect())

[('a', 2), ('b', 1)]

sorted(rdd.groupByKey().mapValues(list).collect())

[('a', [1, 1]), ('b', [1])]

**RDD.flatMap(func: Callable[[T], Iterable[U]], preservesPartitioning: bool = False)**

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

**Exampple:**

rdd = sc.parallelize([2, 3, 4])

print(rdd.flatMap(lambda x: range(1, x)).collect())

**#Output : - [1, 1, 2, 1, 2, 3]**

**RDD.map(func: Callable[[T], U], preservesPartitioning: bool = False)**

Return a new RDD by applying a function to each element of this RDD.

**Example**:

rdd = sc.parallelize(["b", "a", "c"])

sorted(rdd.map(lambda x: (x, 1)).collect())

**#Output: - [('a', 1), ('b', 1), ('c', 1)]**

**Example: --- good**

Change the one of the column data into list.

**Solution:**

#read the csv file into pyspark dataframe

df=spark.read.csv('D:\\PySpark\\Prac1\\iris\\iris.csv',header=True)

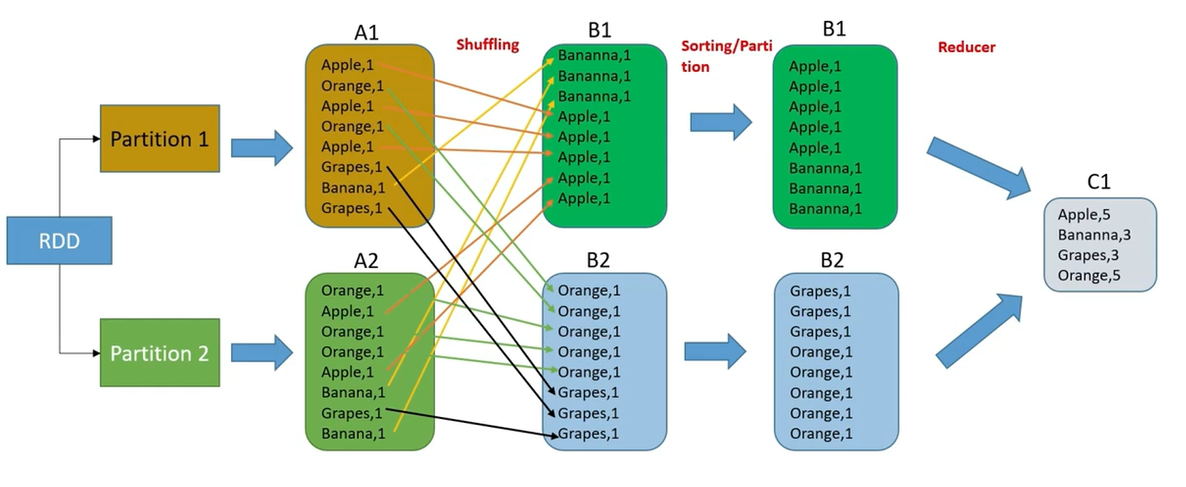
#chaneg the dataframe into rdd and then into list

x=df.select('Sepal\_Width').rdd.flatMap(lambda x: x).collect()

**# Shuffling and Combiner #**

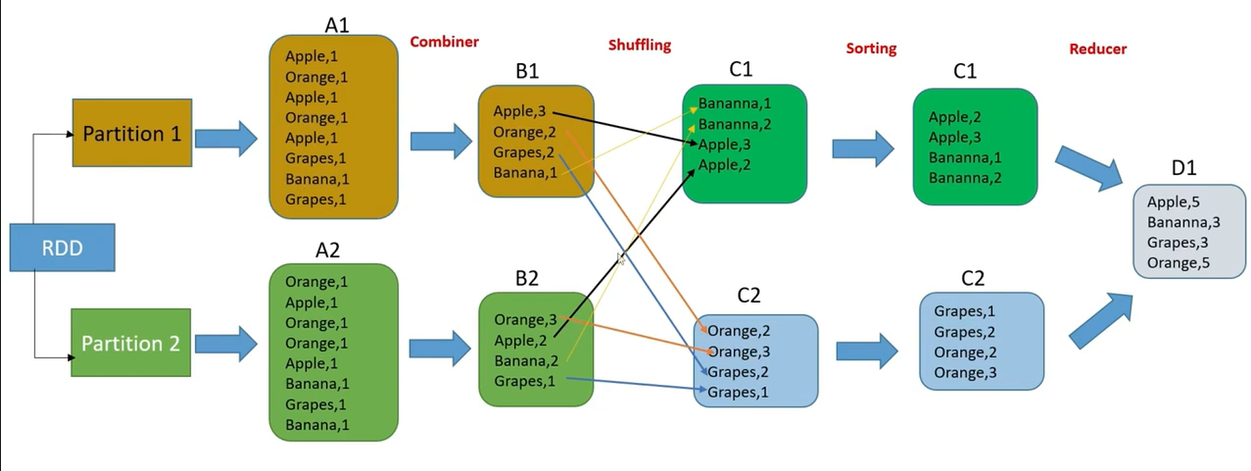
* Shuffling is a process of redistributing data across partitions or even nodes.
* Shuffling operation create a new stage.
* Based on data size we may reduce or increase the number of partitions using configuration spark.sql.shuffle.partitions or through codes like repartition and coalesce.
* Costly operation as it invokes disk I/O and data serialization/de-serialization.
* Spark shuffling triggers for transformation operations like - groupByKey((), reduceByKey(), join, union(), cogroup(), groupBy() etc.
* distinct creates a shuffle.
* Count and CountByKey does not create any shuffle.
* Avoid shuffling at all cost, if Shuffling is absolutely necessary use combiner.
* Out of 3 main key aggregation APIs , the groupByKey doesn't use a combiner and so should be avoided.
* The reduceByKey and aggregateByKey use the combiner and should be preferred.

**Shuffling without combiner**



We can see here that shuffling is causing heavy data operations and flow if used without combiner as compared to shuffling with combiner (see below picture)

**Shuffling with combiner**



**################**

**# Key Aggregation #**

**################**

For Key Aggregation we have below methods-

* RDD.groupByKey(numPartitions=None, partitionFunc=<function portable\_hash>)
  + **Does not uses combiner**
  + can be used for aggregation but should be given low priority as it doesn't use the combiner.
  + when called on dataset of key-value pairs, returns a dataset of (key,iterable<V>) pairs
* RDD.aggregateByKey(zeroValue, seqFunc, combFunc, numPartitions, partitionFunc)
  + Return pyspark.rdd.RDD[Tuple[K, U]].
  + It is used when RDD is in form of key-value pair ( K, V)
  + **Uses combiner**
  + First aggregate elements in each partition and then aggregating the results of all partition to get the final result and then result could be any type than the type of your RDD.
  + 3 mandatory arguments-
  + **zerovalue**: Initial value to initialize the accumulator (used in seqop function). Use 0 for integer and NULL for collection. ( U )
  + **SeqOp**: Function used to accumulate the result of each partition, and store the running accumulated result to U (U,T) => U. a function to merge a V (value ) into a U

Note: Seqop takes atleast two arg—**accumulator as first arggumnet and element as second argument**.(val of key-val of rdd)

* + **CombOp**: Function is used to combine result of all partitions U.

It takes two arguments: two accumulators. The accumulators are from different partitions of the RDD. The combined operation merges them by doing some operation with them. For example, it can add them together or find the maximum.

**Note:**

In seqOp or comOp value from key-value RDD pair is passed.

* RDD.reduceByKey(func, numPartitions= None, partitionFunc)
  + Returns pyspark.rdd.RDD[Tuple[K, V]]
  + **Uses combiner**
  + when called on **a dataet of (K,V) pairs, returns a datset of (K,V)** **pairs where the values for each key are aggregated using the given reduce function**, which must be of type (V,V)=>V.
  + Like in groupByKey, the number of reduce task is configurable through an optinal argument- numPartitions.
  + It uses combiner, associative reduction.
* RDD.countByKey()
  + Returns Dict[K, V], K= key and V = count of that key. Returns result as dictionary
  + **Don’t create any shuffle**

**Question: -- GroupByKey()**

From order\_items file create the RDD using key-value pair. Key is at index 2 and item is at index 4 and then get aggregated sum for each key.

**Solution:**

sc=spark.sparkContext

from datetime import datetime

#D:\Udemy\_PySpark\RetailDB+SalesData\RetailDB SalesData\Orders\part-00000

rdd1=sc.textFile('D:\\Udemy\_PySpark\\RetailDB+SalesData\\RetailDBSalesData\\Order\_items\\ order')

#now get the customer\_id at second index and ordered amount at 4th index in key-value pair

rdd1=rdd1.map(lambda x : (int(x.split(',')[2]),float(x.split(',')[4])) )

#now group by each key

rddgrp=rdd1.groupByKey()

#get the sum for each key

rddgrp=rddgrp.mapValues(sum)

print(rddgrp.take(10))

**Question: -- ReduceByKey()**

**Example 1: -- Revenue for each order\_item**

Find the total revenue sold for each each order\_item id. Assume that order\_item\_id is at index 1 and amount is at index 4.

**Solution:**

from operator import add

sc=spark.sparkContext

#D:\Udemy\_PySpark\RetailDB+SalesData\RetailDB SalesData\Orders\part-00000

rdd1=sc.textFile('D:\\Udemy\_PySpark\\RetailDB+SalesData\\RetailDBSalesData\\Order\_items\\part-00000')

#now get the order\_item\_id at index 1 and ordered amount at index 4 in key-value pair

rdd1=rdd1.map(lambda x : (int(x.split(',')[1]),float(x.split(',')[4])) )

#reduce the rdd using the reduceByKey

rdd\_result=rdd1.reduceByKey(add)

**Example2:**

Find the maximum revenue for each order\_item\_id.

**Solution:**

from operator import add

sc=spark.sparkContext

#D:\Udemy\_PySpark\RetailDB+SalesData\RetailDB SalesData\Orders\part-00000

rdd1=sc.textFile('D:\\Udemy\_PySpark\\RetailDB+SalesData\\RetailDBSalesData\\Order\_items\\part-00000')

#now get the order\_item\_id at index 1 and ordered amount at index 4 in key-value pair

rdd1=rdd1.map(lambda x : (int(x.split(',')[1]),float(x.split(',')[4])) )

#reduce the rdd using the reduceByKey

rdd\_result=rdd1.reduceByKey( lambda x,y : x if x>y else y)

print(rdd\_result.take(10))

**RDD.reduce(func)**

Reduces the elements of this RDD using the specified commutative and associative binary operator. Currently reduces partitions locally.

**RDD.reduce(func)**

**Question:**

From order RDD find the max of order amount for order id=10.

**Solution:**

sc=spark.sparkContext

rdd=sc.textFile("D:\\Udemy\_PySpark\\RetailDB+SalesData\\RetailDBSalesData\\Order\_items\\part-00000")

print(rdd.take(5))

#get the rdd for order id=10

rdd=rdd.filter(lambda x : int(x.split(',')[1])==10)

#now get the reccord for which order amoutn is max

res=rdd.reduce(lambda a,b:a if (float(a.split(',')[4])> float(b.split(',')[4])) else b)

print(res)

**reduce vs reduceByKey vs aggrerate()**

 **reduce()** is similar to [aggregate()](https://sparkbyexamples.com/apache-spark-rdd/spark-rdd-aggregate-function-example/) with a difference; reduce return type should be the same as this RDD element type whereas aggregation can return any type.

 **reduce()** also same as reduceByKey() except reduceByKey() operates on [Pair RDD](https://sparkbyexamples.com/apache-spark-rdd/spark-pair-rdd-functions/)

**#Example on aggregateByKey #**

Let say be have below RDD which we will be using for exercise.

[(2, 'Jose', 200), (2, 'Jimy', 250), (2, 'Tina', 130), (4, 'Jimy', 50), (4, 'Tina', 300), (4, 'Jose', 150), (4, 'Ram', 200), (7, 'Tina', 200), (7, 'Jose', 300), (7, 'Jimy', 200)]

Each element of RDD have three element(id, name, amount) from order file/table

rdd=sc.parallelize([(2,"Jose",200),(2,"Jimy",250),(2,"Tina", 130),(4,"Jimy",50),(4,"Tina",300),

(4,"Jose",150),(4,"Ram",200),(7,"Tina",200),(7,"Jose",300),(7,"Jimy",200)],2)

**Question: --- Good**

Find the order id, name and amount for which order is maximum for each order id group.

**Solution:**

#Create the rdd in key-value pair

rdd=rdd.map(lambda x: (x[0],(x[1],x[2])))

for i in rdd.take(5):

    print(i)

#Now get the id and max of amount

zero\_value=("",0)

#Create the seq\_opration function

#elem is value from each elem

def seq\_op(acc,elem):

    if acc[1]>elem[1]:

        return acc

    else:

        return elem

#Create the comb\_opration function

#elem is value from each elem

def comb\_op(acc1,acc2):

    if acc1[1]>acc2[1]:

        return acc1

    else:

        return acc2

#Now apply seqOp and CombOp function

aggr\_ordItems=rdd.aggregateByKey(zero\_value,seq\_op,comb\_op)

print(aggr\_ordItems.take(5))

**Question: -- good**

Find the sum of amount for each order id and count of each order\_id.

**Solution:**

#Create the rdd in key-value pair

rdd=rdd.map(lambda x: (x[0],(x[1],x[2])))

for i in rdd.take(5):

    print(i)

#Now get the id and max or amount

zero\_value=(0,0)

#Create the seq\_opration function

#elem is value from each elem

def seq\_op(acc,elem):

    return (acc[0]+elem[1],acc[1]+1)

#Create the comb\_opration function

#elem is value from each elem

def comb\_op(acc1,acc2):

    return (acc1[0]+acc2[0],acc1[1]+acc2[1])

#Now apply those function

aggr\_ordItems=rdd.aggregateByKey(zero\_value,seq\_op,comb\_op)

print(aggr\_ordItems.take(3))

##################

# Ranking in Pyspark #

##################

We can divide ranking in two groups –

1. Global ranking
2. Ranking per group

|  |  |
| --- | --- |
|  |  |

**Global Ranking:**

It can be acheived by below function

* sortByKey() and take
* takeOrdered() or top

**Ranking per group**

Getting ranking per group us a bit complex but important tow know

Per-Key or per group ranking can be achieved using-

* groupByKey() with flatmap()
* Python knowledge like sorted function, list etc

**Example: -- Global Ranking --- Using sortByKe()**

Get the top 5 products with heighest price

sc=spark.sparkContext

#D:\Udemy\_PySpark\RetailDB+SalesData\RetailDB SalesData\Orders\part-00000

rdd1=sc.textFile('D:\\Udemy\_PySpark\\RetailDB+SalesData\\RetailDBSalesData\\Products\\product')

#for few producs there are null/no value of price, remove those values

rdd1=rdd1.filter(lambda x : x.split(',')[4]!='')

#now create the paired RDD

rdd1=rdd1.map(lambda x : (float(x.split(',')[4]),x))

#now sortByKey

rdd1=rdd1.sortByKey(ascending=False)

print(rdd1.take(4))

**Example: -- Global Ranking --- Using takeOrdered()**

rdd.takeOrdered(5, lambda x : 1-\*float( x.split(',')[4] ) ) #-- multiplying by -1 to get in descneding order

**Ranking per group**

**Question:** For each category id in [2,3,4] (category is at index 1 in products file) whilethe top two products

rdd=rdd.filter( lambda x : (int(x.split(',')[1]) in [2,3,4] and (int(x.split(',') in [1,2,3,4,5,25,26,27,28,29])) ))

#now create the group for each item - 2,3,4

rdd=rdd.map(lambda x : ( int(x.split(',')[1]),x ).groupByKey()

#now take the first group i.e for value=2

first=rdd.first()

#now get the value for this group

sorted( first[1], key=lambda x : flost(x.split(',')[4],reverse=True) )

###################

# Set TransFomration #

###################

For set transofomration we have below functions/APIs-

1. RDD.union( rdd2, rdd3, . . . . rddn)
2. rdd1.intsersection( rdd2 )
3. rdd1.substract(rdd2) – if rdd2 data are in rdd1 then remove that data from rdd1 and return rdd1

**union(other)**

A union will get all element from both dataset

In case of union , it will not get distinct elements. Apply disticnt, if you want to get distinct elements after union operation.

When we use set operations such as union and intersect, data should hev similar atructure ( same column and types)

**Example:**

Find the number of customers who place order in July or Aug month from order.

**Solution:**

sc=spark.sparkContext

#D:\Udemy\_PySpark\RetailDB+SalesData\RetailDB SalesData\Orders\orders.txt

rdd=sc.textFile('D:\\Udemy\_PySpark\\RetailDB+SalesData\\RetailDBSalesData\\Orders\\order.txt')

print(rdd.count())

#filter rdd data for aug and july month

rdd=rdd.filter(lambda x : (x.split(',')[1].split('-')[1]=='07') | (x.split(',')[1].split('-')[1]=='08'))

print(rdd.count())

**# rdd1.intsersection( rdd2 ) #**

It gives the intersection of two RDDs

Output will not contain any duplicates.

**Question:**

Find the customer id who have order in July and Aug month both, from order file.

**Solution:**

sc=spark.sparkContext

#D:\Udemy\_PySpark\RetailDB+SalesData\RetailDB SalesData\Orders\part-00000

rdd=sc.textFile('D:\\Udemy\_PySpark\\RetailDB+SalesData\\RetailDBSalesData\\Orders\\order.txt')

#ger the customer for july month

rddJuly=rdd.filter(lambda x : (x.split(',')[1].split('-')[1]=='07')).map(lambda x : x.split(',')[2])

print('number of costomer for July month: ',rddJuly.count())

#ger the customer for aug month

rddAug=rdd.filter(lambda x : (x.split(',')[1].split('-')[1]=='08')).map(lambda x : x.split(',')[2])

#ger the common  customer for july and aug month

print('number of costomer for Aug month: ',rddAug.count())

rddCommon=rddJuly.intersection(rddAug)

print('common customer count for aug and july month: ',rddCommon.count())

**######################**

**# Sample Transformation #**

**######################**

**sample(withReplacement=bool, fraction, seed=None)**

To get random sample record from the RDD

with replacement: If True , same result can be prodiced more than once

Fraction: from 0 to 1, .3 means upto 30% record want in sample, but not guaranted

seed: Reproduce the same sample:

**takeSample(withReplacement, num, seed=None)**

same as previous but want the number(num) of sample in result.

**What is partition**

If datasets are huge then can not fit on single node and they have to be partitioned across different node or machine.

Partition in spark is basicallu as atomic chunk of data stored on a node in cluster. They are basic units of parallelism.

One partition cann’t span over multiple machine.

**################################# SparkSession #################################**

**SparkSession:**

It is the entry point for in spark 2.0 ownwards.

Prior to 2.0 SparkContextt was used to be entry point.

* **SparkContext**: Entry point to work with RDD, accumulators and broadcast variables (<2.0 spark)
* **SQLConext**: Used for initializing the functionalities of Spark SQL (< spark 2.0)
* **HiveContext**: Super set of SQLContext (< spark 2.0)

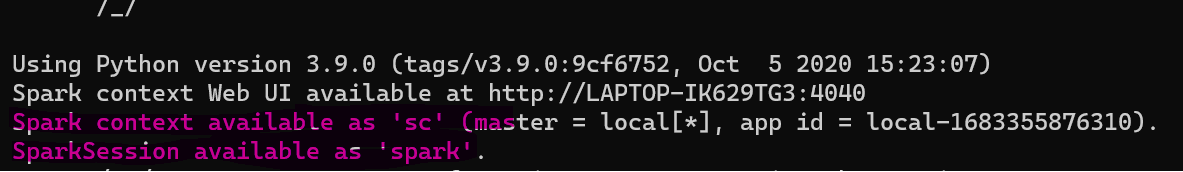
Almost all the APIs available in SparkContext, SQLContext, HiveContext are now available in SparkSession.

PySparkSession

class pyspark.sql.SparkSession(sparkContext, jsparkSession=None)

*It is unified entry point to Spark to work with RDD, DataFrame, and Dataset on/after spark 2.0 version*.

When we open pyspark shell spark by default creates a session by name 'spark'



A SparkSession can be used create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files.

To create a SparkSession, use the following builder pattern:

builder

A class attribute having a Builder to construct SparkSession instances

Example:

spark = SparkSession.builder \

.master("local") \

.appName("Word Count") \

.config("spark.some.config.option", "some-value") \

.getOrCreate()

Example 1:

Read the data from csv file and convert into dataframe.

from pyspark.sql import SparkSession

print('imported')

spark = SparkSession.builder.appName('PySpark Read CSV').getOrCreate()

spark.sparkContext.setLogLevel("ERROR")

df = spark.read.csv("D:\\PySpark\\Prac1\\flights.csv")

print(type(df)) #<class 'pyspark.sql.dataframe.DataFrame'>

print('\*\*\*\*\*\*\*')

**Methods of PySparkSession:**

|  |  |
| --- | --- |
| [SparkSession.builder.appName](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.builder.appName.html#pyspark.sql.SparkSession.builder.appName)(name) | Sets a name for the application, which will be shown in the Spark web UI. |
| [SparkSession.builder.config](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.builder.config.html#pyspark.sql.SparkSession.builder.config)([key, value, conf]) | Sets a config option. |
| [SparkSession.builder.enableHiveSupport](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.builder.enableHiveSupport.html#pyspark.sql.SparkSession.builder.enableHiveSupport)() | Enables Hive support, including connectivity to a persistent Hive metastore, support for Hive SerDes, and Hive user-defined functions. |
| [SparkSession.builder.getOrCreate](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.builder.getOrCreate.html#pyspark.sql.SparkSession.builder.getOrCreate)() | Gets an existing [SparkSession](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.html#pyspark.sql.SparkSession) or, if there is no existing one, creates a new one based on the options set in this builder. |
| [SparkSession.builder.master](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.builder.master.html#pyspark.sql.SparkSession.builder.master)(master) | Sets the Spark master URL to connect to, such as “local” to run locally, “local[4]” to run locally with 4 cores, or “spark://master:7077” to run on a Spark standalone cluster. |
| [SparkSession.catalog](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.catalog.html#pyspark.sql.SparkSession.catalog) | Interface through which the user may create, drop, alter or query underlying databases, tables, functions, etc. |
| [SparkSession.conf](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.conf.html#pyspark.sql.SparkSession.conf) | Runtime configuration interface for Spark. |
| [SparkSession.createDataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.createDataFrame.html#pyspark.sql.SparkSession.createDataFrame)(data[, schema, …]) | Creates a [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) from an RDD, a list or a pandas.DataFrame. |
| [SparkSession.getActiveSession](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.getActiveSession.html#pyspark.sql.SparkSession.getActiveSession)() | Returns the active SparkSession for the current thread, returned by the builder |
| [SparkSession.newSession](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.newSession.html#pyspark.sql.SparkSession.newSession)() | Returns a new SparkSession as new session, that has separate SQLConf, registered temporary views and UDFs, but shared SparkContext and table cache. |
| [SparkSession.range](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.range.html#pyspark.sql.SparkSession.range)(start[, end, step, …]) | Create a [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) with single [pyspark.sql.types.LongType](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.types.LongType.html#pyspark.sql.types.LongType) column named id, containing elements in a range from start to end (exclusive) with step value step. |
| [SparkSession.read](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.read.html#pyspark.sql.SparkSession.read) | Returns a DataFrameReader that can be used to read data in as a [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| [SparkSession.readStream](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.readStream.html#pyspark.sql.SparkSession.readStream) | Returns a DataStreamReader that can be used to read data streams as a streaming [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| [SparkSession.sparkContext](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.sparkContext.html#pyspark.sql.SparkSession.sparkContext) | Returns the underlying SparkContext. |
| [SparkSession.sql](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.sql.html#pyspark.sql.SparkSession.sql)(sqlQuery) | Returns a [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) representing the result of the given query. |
| [SparkSession.stop](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.stop.html#pyspark.sql.SparkSession.stop)() | Stop the underlying SparkContext. |
| [SparkSession.streams](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.streams.html#pyspark.sql.SparkSession.streams) | Returns a StreamingQueryManager that allows managing all the StreamingQuery instances active on this context. |
| [SparkSession.table](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.table.html#pyspark.sql.SparkSession.table)(tableName) | Returns the specified table as a [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| [SparkSession.udf](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.udf.html#pyspark.sql.SparkSession.udf) | Returns a UDFRegistration for UDF registration. |
| [SparkSession.version](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.version.html#pyspark.sql.SparkSession.version) | The version of Spark on which this application is running. |
| [createDataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.createDataFrame.html#pyspark.sql.SparkSession.createDataFrame)(data[, schema, …])  data should be in list/tupe/iterable of iterable(list/tupe/iterable, pyspark Row )  form for each row.  shema = list of col name (no data type) or StructField | Creates a [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) from an RDD, a list or a pandas.DataFrame. |
| [getActiveSession](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.getActiveSession.html#pyspark.sql.SparkSession.getActiveSession)() | Returns the active SparkSession for the current thread, returned by the builder |
| [newSession](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.newSession.html#pyspark.sql.SparkSession.newSession)() | Returns a new SparkSession as new session, that has separate SQLConf, registered temporary views and UDFs, but shared SparkContext and table cache. |
| [range](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.range.html#pyspark.sql.SparkSession.range)(start[, end, step, numPartitions]) | Create a [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) with single [pyspark.sql.types.LongType](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.types.LongType.html#pyspark.sql.types.LongType) column named id, containing elements in a range from start to end (exclusive) with step value step. |
| [sql](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.sql.html#pyspark.sql.SparkSession.sql)(sqlQuery) | Returns a [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) representing the result of the given query. |
| [stop](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.stop.html#pyspark.sql.SparkSession.stop)() | Stop the underlying SparkContext. |
| [table](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.table.html#pyspark.sql.SparkSession.table)(tableName) | Returns the specified table as a [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |

**spark\_session.createDataFrame(data, schema=None, samplingRatio=None, verifySchema=True)**

data --- rdd data, iterable of iterable , pandas dataframe.

schema --- pyspark.sql.types.DataType, str or list, optional

**Example:**

Create the dataframe by reading iris\_site.csv file. First read into rdd and then convert rdd into dataframe.

**Solution:**

#sc=spark context

#spark = pyspark session

rdd = sc.textFile('D:\\PySpark\\Prac1\\iris\\iris\_site.csv')

rdd=rdd.map(lambda x : x.split(','))

print(type(rdd))#<class 'pyspark.rdd.RDD'>

df=spark.createDataFrame(rdd)

print(type(df))#class 'pyspark.sql.dataframe.DataFrame'>

**Attributes:**

|  |  |
| --- | --- |
| [builder](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.html#pyspark.sql.SparkSession.builder) | A class attribute having a Builder to construct [SparkSession](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.html#pyspark.sql.SparkSession) instances. |
| [catalog](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.catalog.html#pyspark.sql.SparkSession.catalog) | Interface through which the user may create, drop, alter or query underlying databases, tables, functions, etc. |
| [conf](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.conf.html#pyspark.sql.SparkSession.conf) | Runtime configuration interface for Spark. |
| [read](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.read.html#pyspark.sql.SparkSession.read) | Returns a DataFrameReader that can be used to read data in as a [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| [readStream](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.readStream.html#pyspark.sql.SparkSession.readStream) | Returns a DataStreamReader that can be used to read data streams as a streaming [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| [sparkContext](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.sparkContext.html#pyspark.sql.SparkSession.sparkContext) | Returns the underlying SparkContext. |
| [streams](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.streams.html#pyspark.sql.SparkSession.streams) | Returns a StreamingQueryManager that allows managing all the StreamingQuery instances active on *this* context. |
| [udf](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.udf.html#pyspark.sql.SparkSession.udf) | Returns a UDFRegistration for UDF registration. |
| [version](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.version.html#pyspark.sql.SparkSession.version) | The version of Spark on which this application is running. |

**Creating pyspark.sql.DataFrame from csv file:**

For creating dataframe we need to create the session and then call 'read' methods which retrun DatFrameReader. Then from DatFrameReader class we can use 'csv()' to read csv file and create dataframe.

Example:

spark = SparkSession.builder.appName('PySpark Read CSV').getOrCreate()

df = spark.read.csv("D:\\PySpark\\Prac1\\flights.csv")

**################################**

**#pyspark.sql.DataFrameReader class#**

**################################**

Interface used to load a DataFrame from external storage systems (e.g. file systems, key-value stores, etc). Use SparkSession.read to access this

**Method of DatFrameReader class:**

|  |  |
| --- | --- |
| [csv](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.csv.html#pyspark.sql.DataFrameReader.csv)(path[, schema, sep, encoding, quote, …]) | Loads a CSV file and returns the result as a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| [format](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.format.html#pyspark.sql.DataFrameReader.format)(source) | Specifies the input data source format of file type. |
| [jdbc](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.jdbc.html#pyspark.sql.DataFrameReader.jdbc)(url, table[, column, lowerBound, …])  **Example**:  df=spark.read.jdbc(url=  "jdbc:mysql://localhost:3306/learn",  table = "products",  properties={'user':"root",'password':"admin"})  Note: Here we need to provide useid and password as dict for parameter – properties. | Construct a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) representing the database table named table accessible via JDBC URL url and connection properties.  Check example on below link  <https://sparkbyexamples.com/pyspark/pyspark-query-database-table-using-jdbc/> |
| [json](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.json.html#pyspark.sql.DataFrameReader.json)(path[, schema, primitivesAsString, …]) | Loads JSON files and returns the results as a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| [load](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.load.html#pyspark.sql.DataFrameReader.load)([path, format, schema]) | Loads data from a data source and returns it as a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| schema – S**tructType or string** |  |
| [option](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.option.html#pyspark.sql.DataFrameReader.option)(key, value) | Adds an input option for the underlying data source. |
| [options](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.options.html#pyspark.sql.DataFrameReader.options)(\*\*options) | Adds input options for the underlying data source. |
| [orc](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.orc.html#pyspark.sql.DataFrameReader.orc)(path[, mergeSchema, pathGlobFilter, …]) | Loads ORC files, returning the result as a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| [parquet](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.parquet.html#pyspark.sql.DataFrameReader.parquet)(\*paths, \*\*options) | Loads Parquet files, returning the result as a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| [schema](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.schema.html#pyspark.sql.DataFrameReader.schema)(schema) | Specifies the input schema. |
| [table](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.table.html#pyspark.sql.DataFrameReader.table)(tableName) | Returns the specified table as a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| [text](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameReader.text.html#pyspark.sql.DataFrameReader.text)(paths[, wholetext, lineSep, …]) | Loads text files and returns a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) whose schema starts with a string column named “value”, and followed by partitioned columns if there are any. |

**DataFrameReader.csv**(**path, schema, sep, header, dateFormat, timestampFormat, maxColumns**)

**Parameters**

**path : str or list**

string, or list of strings, for input path(s), or RDD of Strings storing CSV rows.

**schema : pyspark.sql.types.StructType or str, optional**

an optional pyspark.sql.types.StructType for the input schema or a DDL-formatted string

(For example col0 INT, col1 DOUBLE).

**Sep : str, optional**

sets a separator (one or more characters) for each field and value. If None is set, it uses the

default value.

**header : str or bool, optional**

uses the first line as names of columns. If None is set, it uses the default value, false.

**ignoreLeadingWhiteSpaces : tr or bool, optional**

A flag indicating whether or not leading whitespaces from values being read should be skipped. If None is set, it uses the default value, false.

**ignoreTrailingWhiteSpace : str or bool, optional**

A flag indicating whether or not trailing whitespaces from values being read should be skipped. If None is set, it uses the default value, false.

**dateFormat : str, optional**

sets the string that indicates a date format. Custom date formats follow the formats at datetime pattern. # noqa This applies to date type. If None is set, it uses the default value, yyyy-MM-dd.

**timestampFormat : str, optional**

sets the string that indicates a timestamp format. Custom date formats follow the formats at datetime pattern. # noqa This applies to timestamp type. If None is set, it uses the default value, yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX].

**####################**

**# Defining StructType: #**

**####################**

While creating dataframe from csv or any file we can define the datatype of each column.

schema = StructType([ \

StructField("col\_name1",col\_data\_type,nullable),

StructField("col\_name2",col\_data\_type,nullable),

.

.

StructField("col\_name",col\_data\_type,nullable)

])

**----- Synytax**

**Parameters are:**

* **col\_name** ---- column name from file --- mandatory
* **col\_data\_type** ---- data type of column from pyspark.sql.types --- mandatory
* **nullable= Boolean** ---- to identify if column is nullable ---- optional

**Note:**

* StructType will be a list of StructField.
* StructField will contain the column name, datatype and nullable value.

**Question:**

Read the iris.csv file into dataframe and also specify the schema type.

**Solution:**

from pyspark.sql import SparkSession

from pyspark.sql.types import StructType,StructField, StringType, IntegerType,FloatType

#create session and context

spark = SparkSession.builder.appName('PySpark Read CSV').getOrCreate()

#define the schema

schema = StructType([ \

    StructField("Sepal\_Length",FloatType(),True), \

    StructField("Sepal\_Width",FloatType(),True), \

    StructField("Petal\_W", FloatType(), True), \

    StructField("Petal\_Width", StringType(), True), \

    StructField("Species", StringType(), True), \

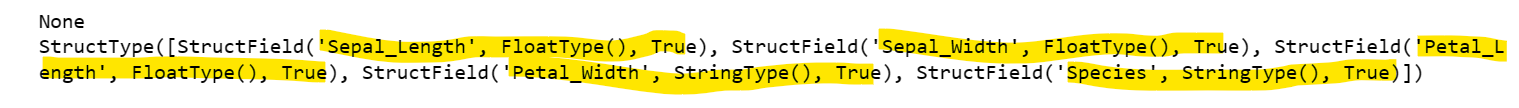
  ])

#create datafraem and assing the schema

df = park.read.csv(path='D:\\PySpark\\Prac1\\iris\\iris.csv',sep=',',header=True,schema=schema)

print(df.schema)

**Output:**



**DataFrameReader.json(path, schema, dropFieldIfAllNull, lineSep, timestampFormat, dateFormat)**

**Schema :** [pyspark.sql.types.StructType](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.types.StructType.html#pyspark.sql.types.StructType) or str, optional

an optional [pyspark.sql.types.StructType](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.types.StructType.html#pyspark.sql.types.StructType) for the input schema or a DDL-formatted string (For example col0 INT, col1 DOUBLE).

**dropFieldIfAllNull :** str or bool, optional

whether to ignore column of all null values or empty array/struct during schema inference. If None is set, it uses the default value, false

Example:

df1 = spark.read.json('python/test\_support/sql/people.json')

**DataFrameReader.format(source: str)**

Specifies the input data source format.

**Example**:

df = spark.read.format('json').load('python/test\_support/sql/people.json')

df.dtypes

[('age', 'bigint'), ('name', 'string')]

**DataFrameReader.load(path: Union[str, List[str], None] = None, format: Optional[str] = None, schema: Union[pyspark.sql.types.StructType, str, None] = None, \*\*options: OptionalPrimitiveType)**

Loads data from a data source and returns it as a DataFrame.

**Parameters**

**path**str or list, optional

optional string or a list of string for file-system backed data sources.

**format**str, optional

optional string for format of the data source. Default to ‘parquet’.

**schema**[pyspark.sql.types.StructType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.StructType.html#pyspark.sql.types.StructType) or str, optional

optional [pyspark.sql.types.StructType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.StructType.html#pyspark.sql.types.StructType) for the input schema or a DDL-formatted string (For example col0 INT, col1 DOUBLE).

**\*\*options**dict

all other string options

**Example: --- Reading json file and convert into dataframe.**

df = spark.read.format('json').load(['python/test\_support/sql/people.json',

'python/test\_support/sql/people1.json'])

df.dtypes

[('age', 'bigint'), ('aka', 'string'), ('name', 'string')]

**Example2**: Read the **csv file into pyspark dataframe**

**Solution:**

df=spark.read.format('csv').load("D:\\PySpark\\Prac1\\flights.csv",sep=',',header=True)

print(df.show(4))

**dataFrameReader.load(path: Union[str, List[str], None] = None, format: Optional[str] = None, schema: Union[pyspark.sql.types.StructType, str, None] = None, \*\*options: OptionalPrimitiveType)**

Loads data from a data source/file and returns it as a DataFrame.

**path:** str or list, optional, file path

**format:** str, optional -- input source format (csv, xlsx, orc, parquet etc)

**options:** dict, all other operaion parameter ( seperator etc)

**Example 2**: Here we are reading text file which is comma separated.

ord=spark.read.load("D:\\Udemy\_PySpark\\RetailDB+SalesData\\RetailDBSalesData\\Orders\\order",sep=',',format='csv',schema=('order\_id int,order\_date timestamp,customer\_id int,order\_datatus string'))

print(ord.show())

**schema:**

This defined the column names of dataframe that could be StrucType or custom type. We can declare schema type in below two ways-

**Method 1:**

**schema =('col1 col1\_type, col2 col2\_type, . . . , coln coln\_type' )**------ in cols and type in one string.

---- this is applicable for all methods(read.csv, read.orc etc ) for creating dataframe from file.

**schema='a long, b double, c string, d date, e timestamp'** --- col and type comma sep. one string

---- when creating from iterable of iterable

**Method 2:**

schema ='col1, col2, . . . , coln ' ------ just to assign header but no datatype.

**Method 3: --- Always use this in coding**

schema = StructType([

StructField('col\_name', spark.sql.type\_datatype, boolean ),

StructField('col\_name', spark.sql.type\_datatype, boolean ),

.

.

StructField('col\_name', spark.sql.type\_datatype, boolean ),

])

**################################**

**# Creating DataFrame from database #**

**################################**

We can read data from sql database tables, oracle database tables etc and create dataframe.

For reading/creating dataframe from table we have below two ways-

**Method 1:** --- Using table name, select all column from specified table ( dbtable = "table\_name" )

dataframe\_mysql = spark.read.format("jdbc")\

.options(url="jdbc:mysql://localhost:3306/**database\_name**",

dbtable = "**table\_name**",user="**user\_id**",password="**password**").load()

**Method 2:** ---- Selecting data using query

dataframe\_mysql = spark.read.format("jdbc")\

.options(url="jdbc:mysql://localhost:3306/**database\_name**",

dbtable = "**(select\_query) query\_alias\_name**",user="**user\_id**",password="**password**").load()

**Note:**

* url , dbtable , user and password varies query to query and database to database
* url='jdbc:db2://xx.xx.xxx.xxx:50000/myDatabase' --- for DB2
* url="jdbc:mysql://localhost:3306/my\_db\_name" --- for mysql database
* url="jdbc:oracel:thin@xx.xx.xxx.xxx:1521/db\_name" --- for oracel db

**Method 3**: ----- **using jdbc method**

df=spark.read.jdbc(url="jdbc:mysql://localhost:3306/learn",table = "products",properties={'user':"root",'password':"admin"})

Observation:

|  |  |
| --- | --- |
| DataFrameReader. jdbc(url,table=tbl\_nm,  properties={'user':"root",'password':"admin"})   * Userid and password in dict format * table --- name of table to read data not query   Example  df=spark.read.jdbc(url=url,table = "products",  properties={'user':"root",'password':"admin"}) | DataFrameReader.format("jdbc").\  option(url,  **dbtable**,**user\_id=user\_id,**  **password=password**).load()   * user\_id and password in comma   separted format.   * dbtable="(select\_query) query\_alias\_name"   Example:  spark.read.format("jdbc")\  .options(url=,\  dbtable = "(select \* from prodcuts) qr",\  user="user\_id",\  password="password").load() |

**Question:**

Read student table and convert into dataframe.

**Solution:**

dataframe\_mysql = spark.read.format("jdbc")\

.options(url="jdbc:mysql://localhost:3306/learn",

         dbtable = "student",user="root",password="admin").load()

print(dataframe\_mysql.show(5))

**Question:**

Read the name and age column from student table

**Solution:**

dataframe\_mysql = spark.read.format("jdbc")\

.options(url="jdbc:mysql://localhost:3306/learn",

         dbtable = **"(select Name,Age from student) qry**", user="root",password="admin")\

.load()

print(dataframe\_mysql.show(5))

**Note:**

**(select Name,Age from student) qry** --- this alias the select query "select Name,Age from student" by name 'qry'

**## Reading data from table using some join condition ##**

In such case we need to just specify the query with join condition nothing else will change.

df=spark.read.format("jdbc")\

.options(url="jdbc:mysql://localhost:3306/orders",

         dbtable = "(select CUST\_NAME,CUST\_CITY,ORD\_AMOUNT,ORD\_DATE from orders join customer on customer.CUST\_CODE=orders.CUST\_CODE) query\_alias\_name",

         user="root",password="admin").load()

print(df.show())

**################################################################################**

**# DataFrame Writer #**

**################################################################################**

df.write ------ it create the dataframe writer object

|  |  |
| --- | --- |
| [bucketBy](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.bucketBy.html#pyspark.sql.DataFrameWriter.bucketBy)(numBuckets, col, \*cols) | Buckets the output by the given columns. |
| [csv](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.csv.html#pyspark.sql.DataFrameWriter.csv)(path[, mode, compression, sep, quote, …]) | Saves the content of the [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) in CSV format at the specified path. |
| [format](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.format.html#pyspark.sql.DataFrameWriter.format)(source) | Specifies the underlying output data source. |
| [insertInto](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.insertInto.html#pyspark.sql.DataFrameWriter.insertInto)(tableName[, overwrite]) | Inserts the content of the [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) to the specified table. |
| [jdbc](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.jdbc.html#pyspark.sql.DataFrameWriter.jdbc)(url, table[, mode, properties]) | Saves the content of the [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) to an external database table via JDBC. |
| [json](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.json.html#pyspark.sql.DataFrameWriter.json)(path[, mode, compression, dateFormat, …]) | Saves the content of the [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) in JSON format ([JSON Lines text format or newline-delimited JSON](http://jsonlines.org/)) at the specified path. |
| [mode](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.mode.html#pyspark.sql.DataFrameWriter.mode)(saveMode) | Specifies the behavior when data or table already exists. |
| append—append in existing table or view  overwrite – if tbl or view exist then delete it’s data and insert new data  ErrorExist – throws error if tbl/view exist else creates and inserts  Ignore – If tgt exists then skip operation |  |
| [option](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.option.html#pyspark.sql.DataFrameWriter.option)(key, value) | Adds an output option for the underlying data source. |
| [options](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.options.html#pyspark.sql.DataFrameWriter.options)(\*\*options) | Adds output options for the underlying data source. |
| [orc](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.orc.html#pyspark.sql.DataFrameWriter.orc)(path[, mode, partitionBy, compression]) | Saves the content of the [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) in ORC format at the specified path. |
| [parquet](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.parquet.html#pyspark.sql.DataFrameWriter.parquet)(path[, mode, partitionBy, compression]) | Saves the content of the [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) in Parquet format at the specified path. |
| [partitionBy](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.partitionBy.html#pyspark.sql.DataFrameWriter.partitionBy)(\*cols) | Partitions the output by the given columns on the file system. |
| [save](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.save.html#pyspark.sql.DataFrameWriter.save)([path, format, mode, partitionBy]) | Saves the contents of the [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) to a data source. |
| [saveAsTable](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.saveAsTable.html#pyspark.sql.DataFrameWriter.saveAsTable)(name[, format, mode, partitionBy]) | Saves the content of the [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) as the specified table. |
| [sortBy](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.sortBy.html#pyspark.sql.DataFrameWriter.sortBy)(col, \*cols) | Sorts the output in each bucket by the given columns on the file system. |
| [text](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameWriter.text.html#pyspark.sql.DataFrameWriter.text)(path[, compression, lineSep]) | Saves the content of the DataFrame in a text file at the specified path. |

**Question:**

Write a dataframe into mysql table

**Solution:**

df.write.mode('append').format("jdbc").options(url="jdbc:mysql://localhost:3306/kaggle",dbtable = "bank",user="root",password="admin").save()

**Note:**

It will pick the column of table in which insert will be performed from dataframe column name.

**#######################**

**# Spark Session: Catalog() #**

**#######################**

* Catalog is introduced in spark 2.0 which is standard API for accessing metadata in Spark SQL
* this work both for SparkSQL and Hive metadata
* We have below methods in catalog for extraction important information

|  |  |
| --- | --- |
| **DataBase function**   * currentDatabase()   returns current database under session   * listDatabase()   returns all database under sesion   * setCurrentDatabase()   set the database under session | **View Function**   * dropGloabalView() * dropTempView() |

|  |  |
| --- | --- |
| Table Function:   * listColumns() * listTables() * cacheTables() * isCached() * uncacheTable() * clearCache() * recoverPartition() * refreshTable() * refreshByPath() | **Function Based Functions:**  listFunctions()  registerFunction(=spark.udf.register) |

**##########################**

**# Spark Session : newSession #**

**##########################**

* Spark\_session.newSession(), Returns a new SparkSession as new session, that has separate SQLConf , registered temporary views and UDFs, but shared SparkContext and table cache.
* **The udfs registered on one sesision will not be visible to the new session**.

**Example 1: (Using udf ) 🡪 Different SparkContext**

new\_spark= spark.newSession

import string

from pyspark.sql.functions import udf

from pyspark.sql.types import StringType,IntegerType

@udf (returnType= StringType())

def initCap str ):

finalStr=""

ar= str.split (" ")

for word in ar:

finalStr= finalStr + word(0:1).upper() + word(1:len(word)) + "

return string.strip finalStr

**Accessing on same session:**

spark.udf.register("initcap1")

spark.sql(""" select emp\_name **, initCap (emp\_name )** from default.emp """).show()

**Accessing on different session:**

new\_spark.sql(""" select emp\_name , **initCap (emp\_name )** from default.emp """).show() **#udf not accessible**

**Example 2: (Using Table data) -🡪Shared SparkContext**

new\_spark= spark.newSession

spark.sql(""" create table student(name int )

spark.sql(""" insert into student values (1)

spark.sql(""" select count(\*) from student """).

new\_spark.sql(""" select count(\*) from student """).

**################################################################################**

**# Pyspark DataFrame #**

**################################################################################**

Data Frame is an immutable distributed collection of data that is organized into named columns, analogous to a table in a relational database.

When do any operation on dataframe, in backend it is performed on RDD.

we can create Data Frames from an existing RDD, from a Hive table, or from data sources.

**Creating dataframe:**

We can create dataframe in pyspark in below ways-

1. From csv file.
2. From json file.
3. From textfile.
4. Construct a DataFrame representing the database table named table accessible via JDBC URL url and connection properties.
5. From rdd --- spark\_sesson.createDataFrame(rdd\_data)
6. Using list , list of tuple etc

#Example 1

df= spark.createDataFrame( [(‘Hello’,2), (‘GM!’,22), schema=[‘name string’, ‘c2 int’)

Providing schema is optional

#Example 2

df= spark.createDataFrame( [ ‘hello’ 23] ), schema=[‘name string’, ‘c2 int’)

Providing schema is optional

**#Creating from csv file #**

df1 = spark.read.csv(path='./dataset/iris.csv',sep=',')

**# Creating from json file #**

iris1\_df1 = spark.read.json('./dataset/iris.json')

**######################################################**

**#createting dataframe from spark rdd and assigning the schema #**

**######################################################**

We can create the dataframe from RDD using createDataFrame() from spark session and assign the schema also.

**spark\_session.createDataFrame(data, schema=None, samplingRatio=None, verifySchema=True)**

**Parameters are:**

* **data** --- rdd data
* **schema** --- user defined schema type or StrucType

**Question:**

Create a rdd from iris\_site.csv file and then convert that rdd into DataFrame and assign the schema also.

**Solution:**

#create session and context

#sc=spark context

#spark = pyspark session

iris\_schema = pyspark.sql.types.StructType([

StructField("Sepal\_Length", FloatType(), True),

StructField("Sepal\_Width", FloatType(), True),

StructField("Petal\_Length", FloatType(), True),

StructField("Petal\_Width", FloatType(), True),

StructField("Species", StringType(), True)

])

iris1 = sc.textFile("./dataset/iris\_site.csv")

iris1\_split = iris1.map(lambda line: line.split(","))

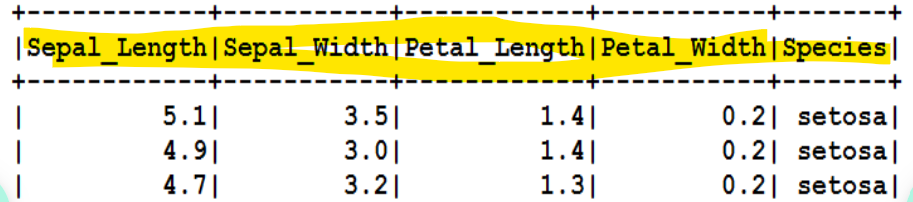
#iris1\_split = iris1\_split.map(lambda var1: [float(var1[0]), float(var1[1]), #float(var1[2]), float(var1[3]),var1[4]])

#creatting dataframe and assigning schema

df1=spark.createDataFrame(iris1\_split, **iris\_schema** )

df1.show()

Output:



**Note:**

This way we can assign the header or column name to dataframe also.

**Attributes of pyspark.sql.DataFrame :**

|  |  |
| --- | --- |
| [columns](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.columns.html#pyspark.sql.DataFrame.columns) | Returns all column names as a list. |
| [dtypes](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.dtypes.html#pyspark.sql.DataFrame.dtypes) | Returns all column names and their data types as a list. |
| [isStreaming](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.isStreaming.html#pyspark.sql.DataFrame.isStreaming) | Returns True if this Dataset contains one or more sources that continuously return data as it arrives. |
| [na](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.na.html#pyspark.sql.DataFrame.na) | Returns a [DataFrameNaFunctions](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrameNaFunctions.html#pyspark.sql.DataFrameNaFunctions) for handling missing values. |
| [rdd](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.rdd.html#pyspark.sql.DataFrame.rdd) | Returns the content as an [pyspark.RDD](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.RDD.html#pyspark.RDD) of [Row](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.Row.html#pyspark.sql.Row). |
| [schema](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.schema.html#pyspark.sql.DataFrame.schema) | Returns the schema of this [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) as a [pyspark.sql.types.StructType](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.types.StructType.html#pyspark.sql.types.StructType). |
| [stat](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.stat.html#pyspark.sql.DataFrame.stat) | Returns a [DataFrameStatFunctions](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrameStatFunctions.html#pyspark.sql.DataFrameStatFunctions) for statistic functions. |
| [storageLevel](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.storageLevel.html#pyspark.sql.DataFrame.storageLevel) | Get the [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame)’s current storage level. |
| [write](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.write.html#pyspark.sql.DataFrame.write) | Interface for saving the content of the non-streaming [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) out into external storage. |
| [writeStream](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.writeStream.html#pyspark.sql.DataFrame.writeStream) | Interface for saving the content of the streaming [DataFrame](https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) out into external storage. |

**Methods:**

We have huge list of methods for pyspark dataframe, they are **–**

|  |  |
| --- | --- |
| [df.agg(\*exprs)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.agg.html#pyspark.sql.DataFrame.agg)  df.agg ( {'col\_name' : 'agg\_func\_name1', . . })  df.agg( aggregation\_function1,. . ) | [Aggregate on the entire df without groups (shorthand for df.groupBy().agg()).](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.alias(alias)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.alias.html#pyspark.sql.DataFrame.alias) | [Returns a new df with an alias set.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.approxQuantile(col, probabilities, …)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.approxQuantile.html#pyspark.sql.DataFrame.approxQuantile) | [Calculates the approximate quantiles of numerical columns of a df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.cache()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.cache.html#pyspark.sql.DataFrame.cache) | [Persists the df with the default storage level (MEMORY\_AND\_DISK).](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.checkpoint([eager])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.checkpoint.html#pyspark.sql.DataFrame.checkpoint) | [Returns a checkpointed version of this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.coalesce(numPartitions)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.coalesce.html#pyspark.sql.DataFrame.coalesce) | [Returns a new df that has exactly numPartitions partitions.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.colRegex(colName)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.colRegex.html#pyspark.sql.DataFrame.colRegex) | [Selects column based on the column name specified as a regex and returns it as Column.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) |
| [df.collect()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.collect.html#pyspark.sql.DataFrame.collect)  Returns all the records as a list of Row.  Take(n) also returns as list of Row but only first n records. **Check Example 37** | [Returns all the records as a list of Row.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Row.html#pyspark.sql.Row) |
| [df.columns](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.columns.html#pyspark.sql.DataFrame.columns) | Returns all column names as a list. |
| [df.corr(col1, col2[, method])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.corr.html#pyspark.sql.DataFrame.corr) | [Calculates the correlation of two columns of a df as a double value.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.count()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.count.html#pyspark.sql.DataFrame.count) | [Returns the number of rows in this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.cov(col1, col2)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.cov.html#pyspark.sql.DataFrame.cov) | Calculate the sample covariance for the given columns, specified by their names, as a double value. |
| [df.createGlobalTempView(name)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.createGlobalTempView.html#pyspark.sql.DataFrame.createGlobalTempView) | [Creates a global temporary view with this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.createOrReplaceGlobalTempView(name)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.createOrReplaceGlobalTempView.html#pyspark.sql.DataFrame.createOrReplaceGlobalTempView) | Creates or replaces a global temporary view using the given name. |
| [df.createOrReplaceTempView(name)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.createOrReplaceTempView.html#pyspark.sql.DataFrame.createOrReplaceTempView) | [Creates or replaces a local temporary view with this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.createTempView(name)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.createTempView.html#pyspark.sql.DataFrame.createTempView) | [Creates a local temporary view with this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.crossJoin(other)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.crossJoin.html#pyspark.sql.DataFrame.crossJoin) | [Returns the cartesian product with another df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.crosstab(col1, col2)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.crosstab.html#pyspark.sql.DataFrame.crosstab) | Computes a pair-wise frequency table of the given columns. |
| [df.cube(\*cols)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.cube.html#pyspark.sql.DataFrame.cube) | [Create a multi-dimensional cube for the current df using the specified columns, so we can run aggregations on them.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.describe(\*cols)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.describe.html#pyspark.sql.DataFrame.describe)  creates new dataframe for each col by calculating  count,mean,stddev, min and max | Computes basic statistics for numeric and string columns. |
| [df.distinct()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.distinct.html#pyspark.sql.DataFrame.distinct) | Returns a new df containing the distinct rows in this df. |
| [df.drop(\*cols)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.drop.html#pyspark.sql.DataFrame.drop) | [Returns a new df that drops the specified column.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.dropDuplicates([subset])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.dropDuplicates.html#pyspark.sql.DataFrame.dropDuplicates) | [Return a new df with duplicate rows removed, optionally only considering certain columns.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.drop\_duplicates([subset])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.drop_duplicates.html#pyspark.sql.DataFrame.drop_duplicates) | drop\_duplicates() is an alias for dropDuplicates(). |
| [df.dropna([how, thresh, subset])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.dropna.html#pyspark.sql.DataFrame.dropna) | [Returns a new df omitting rows with null values.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.dtypes](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.dtypes.html#pyspark.sql.DataFrame.dtypes) | Returns all column names and their data types as a list. |
| [df.exceptAll(df2)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.exceptAll.html#pyspark.sql.DataFrame.exceptAll)  rows of df1 not in df2 by preserving duplicate  resolves columns by position (not by name) | Return a new df containing rows in this df but not in another df while preserving duplicates. |
| [df.explain([extended, mode])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.explain.html#pyspark.sql.DataFrame.explain) | Prints the (logical and physical) plans to the console for debugging purpose. |
| [df.fillna(value[, subset])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.fillna.html#pyspark.sql.DataFrame.fillna)  subset=\_list\_of\_columns\_names | Replace null values, alias for na.fill().  If column names are not specified with subset then replaces for all column in dataframe |
| [df.filter(condition)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.filter.html#pyspark.sql.DataFrame.filter) | Filters rows using the given condition. |
| [df.first()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.first.html#pyspark.sql.DataFrame.first)  returns **the first row of df as pyspark Row obj.** ( it’s row obj not list of row obj like collect() )  **Check Example:**  **Replace the null by average value** | [Returns the first row as a Row.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Row.html#pyspark.sql.Row) |
| [df.foreach(f)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.foreach.html#pyspark.sql.DataFrame.foreach) | Applies the f function to all Row of this df. |
| [df.foreachPartition(f)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.foreachPartition.html#pyspark.sql.DataFrame.foreachPartition) | [Applies the f function to each partition of this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.freqItems(cols[, support])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.freqItems.html#pyspark.sql.DataFrame.freqItems) | Finding frequent items for columns, possibly with false positives. |
| [df.groupBy(\*cols)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.groupBy.html#pyspark.sql.DataFrame.groupBy) | [Groups the df using the specified columns, so we can run aggregation on them.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.head([n])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.head.html#pyspark.sql.DataFrame.head) | Returns the first n rows. |
| [df.hint(name, \*parameters)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.hint.html#pyspark.sql.DataFrame.hint) | [Specifies some hint on the current df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.inputFiles()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.inputFiles.html#pyspark.sql.DataFrame.inputFiles) | [Returns a best-effort snapshot of the files that compose this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.intersect(other)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.intersect.html#pyspark.sql.DataFrame.intersect) | Return a new df containing rows only in both this df and another df. |
| [df.intersectAll(other)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.intersectAll.html#pyspark.sql.DataFrame.intersectAll) | Return a new df containing rows in both this df and another df while preserving duplicates. |
| [df.isEmpty()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.isEmpty.html#pyspark.sql.DataFrame.isEmpty) | [Returns True if this df is empty.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.isLocal()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.isLocal.html#pyspark.sql.DataFrame.isLocal) | Returns True if the collect() and take() methods can be run locally (without any Spark executors). |
| [df.isStreaming](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.isStreaming.html#pyspark.sql.DataFrame.isStreaming) | [Returns True if this df contains one or more sources that continuously return data as it arrives.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.join(other[, on, how])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.join.html#pyspark.sql.DataFrame.join) | [Joins with another df, using the given join expression.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.limit(num)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.limit.html#pyspark.sql.DataFrame.limit)  return first num records of dataframe  as new dataframe | Limits the result count to the number specified. |
| [df.localCheckpoint([eager])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.localCheckpoint.html#pyspark.sql.DataFrame.localCheckpoint) | [Returns a locally checkpointed version of this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.mapInPandas(func, schema)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.mapInPandas.html#pyspark.sql.DataFrame.mapInPandas) | Maps an iterator of batches in the current df using a Python native function that takes and outputs a pandas df, and returns the result as a df. |
| [df.mapInArrow(func, schema)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.mapInArrow.html#pyspark.sql.DataFrame.mapInArrow) | Maps an iterator of batches in the current df using a Python native function that takes and outputs a PyArrow’s RecordBatch, and returns the result as a df. |
| [df.na](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.na.html#pyspark.sql.DataFrame.na) | [Returns a dfNaFunctions for handling missing values.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameNaFunctions.html#pyspark.sql.DataFrameNaFunctions) |
| [df.observe(observation, \*exprs)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.observe.html#pyspark.sql.DataFrame.observe) | [Observe (named) metrics through an Observation instance.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Observation.html#pyspark.sql.Observation) |
| [**df.orderBy([col1,col2. . . ], ascending=[0 or 1 comma seperated ] \*\*kwargs)**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.orderBy.html#pyspark.sql.DataFrame.orderBy)  sorts the dataframe by given col name  same as –  df.sort( [col1,col2. . .], ascending=[0 or 1 comm seperated ] \*\*kwargs) | [Returns a new df sorted by the specified column(s).](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.persist([storageLevel])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.persist.html#pyspark.sql.DataFrame.persist) | [Sets the storage level to persist the contents of the df across operations after the first time it is computed.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.printSchema()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.printSchema.html#pyspark.sql.DataFrame.printSchema) | Prints out the schema in the tree format. |
| [df.randomSplit(weights[, seed])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.randomSplit.html#pyspark.sql.DataFrame.randomSplit) | [Randomly splits this df with the provided weights.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.rdd](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.rdd.html#pyspark.sql.DataFrame.rdd) | Returns the content as an pyspark.RDD of Row. |
| [df.registerTempTable(name)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.registerTempTable.html#pyspark.sql.DataFrame.registerTempTable) | [Registers this df as a temporary table using the given name.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.repartition(numPartitions, \*cols)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.repartition.html#pyspark.sql.DataFrame.repartition) | [Returns a new df partitioned by the given partitioning expressions.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.repartitionByRange(numPartitions, …)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.repartitionByRange.html#pyspark.sql.DataFrame.repartitionByRange) | [Returns a new df partitioned by the given partitioning expressions.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.replace(to\_replace[, value, subset])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.replace.html#pyspark.sql.DataFrame.replace) | [Returns a new df replacing a value with another value.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.rollup(\*cols)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.rollup.html#pyspark.sql.DataFrame.rollup) | [Create a multi-dimensional rollup for the current df using the specified columns, so we can run aggregation on them.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.sameSemantics(other)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.sameSemantics.html#pyspark.sql.DataFrame.sameSemantics) | [Returns True when the logical query plans inside both dfs are equal and therefore return same results.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.sample([withReplacement, …])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.sample.html#pyspark.sql.DataFrame.sample) | [Returns a sampled subset of this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.sampleBy(col, fractions[, seed])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.sampleBy.html#pyspark.sql.DataFrame.sampleBy) | Returns a stratified sample without replacement based on the fraction given on each stratum. |
| [df.schema](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.schema.html#pyspark.sql.DataFrame.schema) | Returns the schema of this df as a pyspark.sql.types.StructType. |
| [df.select(\*cols)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.select.html#pyspark.sql.DataFrame.select) | [Projects a set of expressions and returns a new df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.selectExpr(\*expr)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.selectExpr.html#pyspark.sql.DataFrame.selectExpr) | [Projects a set of SQL expressions and returns a new df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.semanticHash()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.semanticHash.html#pyspark.sql.DataFrame.semanticHash) | [Returns a hash code of the logical query plan against this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.show([n, truncate, vertical])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.show.html#pyspark.sql.DataFrame.show) | Prints the first n rows to the console. |
| [df.sort(\*cols, \*\*kwargs)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.sort.html#pyspark.sql.DataFrame.sort) | [Returns a new df sorted by the specified column(s).](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.sortWithinPartitions(\*cols, \*\*kwargs)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.sortWithinPartitions.html#pyspark.sql.DataFrame.sortWithinPartitions) | [Returns a new df with each partition sorted by the specified column(s).](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.sparkSession](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.sparkSession.html#pyspark.sql.DataFrame.sparkSession) | [Returns Spark session that created this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.stat](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.stat.html#pyspark.sql.DataFrame.stat) | [Returns a dfStatFunctions for statistic functions.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameStatFunctions.html#pyspark.sql.DataFrameStatFunctions) |
| [df.storageLevel](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.storageLevel.html#pyspark.sql.DataFrame.storageLevel) | [Get the df’s current storage level.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.subtract(other)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.subtract.html#pyspark.sql.DataFrame.subtract) | Return a new df containing rows in this df but not in another df. |
| [df.summary(\*statistics)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.summary.html#pyspark.sql.DataFrame.summary) | Computes specified statistics for numeric and string columns. |
| [df.tail(num)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.tail.html#pyspark.sql.DataFrame.tail) | [Returns the last num rows as a list of Row.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Row.html#pyspark.sql.Row) |
| [df.take(num)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.take.html#pyspark.sql.DataFrame.take)  Returns the first num rows as a list of Row(pyspark.sql.Row). check **Example 37** | [Returns the first num rows as a list of Row.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Row.html#pyspark.sql.Row) |
| [**df.toDF(\*cols)**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.toDF.html#pyspark.sql.DataFrame.toDF) **/ df.toDF(['col1',…'coln'])**  this is for returning the dataframe with new col name.  new col\_name count must be same as actual col name count.  #Example# -- keep first ocuurance and delete other occurance of col from df for duplicate col name | [Returns a new df that with new specified column names](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.toJSON([use\_unicode])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.toJSON.html#pyspark.sql.DataFrame.toJSON) | [Converts a df into a RDD of string.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.toLocalIterator([prefetchPartitions])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.toLocalIterator.html#pyspark.sql.DataFrame.toLocalIterator) | [Returns an iterator that contains all of the rows in this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.toPandas()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.toPandas.html#pyspark.sql.DataFrame.toPandas) | [Returns the contents of this df as Pandas pandas.df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.to\_pandas\_on\_spark([index\_col])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.to_pandas_on_spark.html#pyspark.sql.DataFrame.to_pandas_on_spark) |  |
| [df.transform(func, \*args, \*\*kwargs)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.transform.html#pyspark.sql.DataFrame.transform)   * func – Custom function to call. * \*args – Arguments to pass to func. * \*kwargs – Keyword arguments to pass to func.   **Note:**  We do have same method in spark.sql.functions but it **applies the transformation on column which holds Array type data.** | Returns a new df after applying the transformation.  Check below two example in this doc   * dataframe transorm * functions transform |
| [df.union(other)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.union.html#pyspark.sql.DataFrame.union) | Return a new df containing union of rows in this and another df. |
| [df.unionAll(other)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.unionAll.html#pyspark.sql.DataFrame.unionAll) | Return a new df containing union of rows in this and another df. |
| [df.unionByName(other[, …])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.unionByName.html#pyspark.sql.DataFrame.unionByName)   * in both df **col name should be same (order does not matter)** else error * Can be used to stack vertically two dataframe * **Check example vertical stack** | Returns a new df containing union of rows in this and another df.  **It returns the duplicate rows also like unionAll** |
| [df.unpersist([blocking])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.unpersist.html#pyspark.sql.DataFrame.unpersist) | [Marks the df as non-persistent, and remove all blocks for it from memory and disk.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.where(condition)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.where.html#pyspark.sql.DataFrame.where) | where() is an alias for filter(). |
| [df.withColumn(colName, col)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.withColumn.html#pyspark.sql.DataFrame.withColumn) | [Returns a new df by adding a column or replacing the existing column that has the same name.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.withColumns(\*colsMap)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.withColumns.html#pyspark.sql.DataFrame.withColumns) | [Returns a new df by adding multiple columns or replacing the existing columns that has the same names.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.withColumnRenamed(existing, new)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.withColumnRenamed.html#pyspark.sql.DataFrame.withColumnRenamed) | [Returns a new df by renaming an existing column.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.withMetadata(columnName, metadata)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.withMetadata.html#pyspark.sql.DataFrame.withMetadata) | [Returns a new df by updating an existing column with metadata.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.withWatermark(eventTime, …)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.withWatermark.html#pyspark.sql.DataFrame.withWatermark) | [Defines an event time watermark for this df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.write](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.write.html#pyspark.sql.DataFrame.write) | [Interface for saving the content of the non-streaming df out into external storage.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.writeStream](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.writeStream.html#pyspark.sql.DataFrame.writeStream) | [Interface for saving the content of the streaming df out into external storage.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.writeTo(table)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.writeTo.html#pyspark.sql.DataFrame.writeTo) | Create a write configuration builder for v2 sources. |
| [df.pandas\_api([index\_col])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.pandas_api.html#pyspark.sql.DataFrame.pandas_api) | Converts the existing df into a pandas-on-Spark df. |
| [df.NaFunctions.drop([how, thresh, subset])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameNaFunctions.drop.html#pyspark.sql.DataFrameNaFunctions.drop) | [Returns a new df omitting rows with null values.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.NaFunctions.fill(value[, subset])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameNaFunctions.fill.html#pyspark.sql.DataFrameNaFunctions.fill) | Replace null values, alias for na.fill(). |
| [df.NaFunctions.replace(to\_replace[, …])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameNaFunctions.replace.html#pyspark.sql.DataFrameNaFunctions.replace) | [Returns a new df replacing a value with another value.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.StatFunctions.approxQuantile(col, …)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameStatFunctions.approxQuantile.html#pyspark.sql.DataFrameStatFunctions.approxQuantile) | [Calculates the approximate quantiles of numerical columns of a df.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.StatFunctions.corr(col1, col2[, method])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameStatFunctions.corr.html#pyspark.sql.DataFrameStatFunctions.corr) | [Calculates the correlation of two columns of a df as a double value.](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) |
| [df.StatFunctions.cov(col1, col2)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameStatFunctions.cov.html#pyspark.sql.DataFrameStatFunctions.cov) | Calculate the sample covariance for the given columns, specified by their names, as a double value. |
| [df.StatFunctions.crosstab(col1, col2)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameStatFunctions.crosstab.html#pyspark.sql.DataFrameStatFunctions.crosstab) | Computes a pair-wise frequency table of the given columns. |
| [df.StatFunctions.freqItems(cols[, support])](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameStatFunctions.freqItems.html#pyspark.sql.DataFrameStatFunctions.freqItems) | Finding frequent items for columns, possibly with false positives. |
| [df.StatFunctions.sampleBy(col, fractions)](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrameStatFunctions.sampleBy.html#pyspark.sql.DataFrameStatFunctions.sampleBy) | Returns a stratified sample without replacement based on the fraction given on each stratum. |

**# Selecting data from DataFrame #**

**DataFrame.select(\*cols)**

Projects a set of expressions and returns a new DataFrame. We can use it in below difference ways-

1. df.select( df.col\_name1, df.col\_name2 . . . )
2. df.select ( 'col\_name1','col\_name2',. . . . )
3. df.col\_name1+y,'col\_name2'+x,. . . .

Let say we have order file ( comma seperated) and want to load data into data frame

ord=spark.read.load("D:\\Udemy\_PySpark\\RetailDB+SalesData\\RetailDBSalesData\\Orders\\order",sep=',',format='csv',schema=('order\_id int,order\_date timestamp,customer\_id int,order\_datatus string'))

print(ord.show())

**Question1:**

Select the order id column

**Solution:**

ord.select( ord.order\_id ).show(2) -------- **Method1**

ord.select( 'order\_id' ).show(2) -------- **Method2**

**Question:**

Add 10 in each order id

**Solution:**

ord.select( (ord.order\_id+10).alias('order\_id\_plus\_10') )

**Question:**

Select order\_id, order\_status from datafframe.

**Solution:**

ord.select( 'order\_id','order\_status' )

**Question**- select the order\_status column data and convert into lower case.

For converting into lower case park have **inbuilt function – lower(col)**

**Solution:**

ord.select( lower(ord.order\_status) ) -------- **Method 1**

ord.select( lower('order\_status') ) -------- **Method 2**

**selectExpr( \*expr )**

This is a variant of select that accept SQL expression

df.selectExpr( expression (s) )

If we want to use any function available in SQL but not in Spark Built-in function, then we can use selectExpr.

Example:

ord.selectExpr(' substring(order\_date, 1, 4) as ord\_yr') --- Getting order year from ord dataframe

ord.selectExpr( ' order\_id+10 ' ) ---- Adding 10 in order\_id

**withColumn(new\_name , col\_name)**

* **col\_name** --- Column on which some transformation will be applied
* **new\_name ---** new col\_name to which transformaed values will be assigned

This is used to apply a transformation on a selected column and create a new columns in dataframe.

If new column name is same as selected column then it re-assigned the transformed value to same column.

**Question:**

Create a new column by name ord\_ywar which will contains the only year value of order\_year.

**Solution:**

from pyspark.sql.functions import substring

ord.withColumn('ord\_year', substring(ord.order\_date,1,4)).show()

**Note:**

Out of select, selectExpr and withColumn only withColumn function creates a new column in dataframe.

|  |  |
| --- | --- |
| ord.show() | ord.withColumn('ord\_year', substring(ord.order\_date,1,4)).show() |

**DataFrame.withRenamed( col\_name, new\_name )**

This function renames with existing column to new name.

* col\_name --- column that want to rename
* new\_name ---- new column name

**DataFrame.drop(\*cols)**

Returns a new DataFrame by dropping the specified column.

If column(s) are not in dataframe then no operation is performed

* **cols:** str or :class:Column

Column(s) that want to drop from dataframe

* **axis:** {0 or ‘index’, 1 or ‘columns’}, default 0

|  |  |
| --- | --- |
| Deleting single column | Deleting multiple column |
| * df.drop( 'col\_name' ) * df.drop( df.col\_name ) * df.drop( col( 'col\_name' ) ) | * df.drop( 'col1', 'col2',..'coln' )   cols = ("col1","col2",.."coln")   * df.drop( \*cols ) |

df.drop( **col( 'InvoiceDate' , 'Country' )** )

---- This will give error because col function takes only one column name

Example1:

df.drop('age') --- delete the age column from dataframe

df.drop(df.age) --- delete the age column from dataframe

df.drop(col('age')) --- delete the age column from dataframe

Example2:

df.drop("firstname","middlename","lastname")

Example3:

cols = ("firstname","middlename","lastname")

df.drop(\*cols)

**DataFrame.dropna(how='any', thresh=None, subset=None)**

Returns a new DataFrame omitting rows with null values in rows.. DataFrame.dropna() and DataFrameNaFunctions.drop() are aliases of each other

This method drops rows based on condition for which value is null.

**Parameters are:**

* how: 'any' or 'all'
* thresh: int,optional
* subset: str,tuple or list, optional

**dataframe\_name.dropDuplicates(Column\_name (s) )**

This function is used to remove the duplicated rows from selected column keeping first occurance.

* Col\_name --- Column from which duplicated will be removed.

If no name is specified then it considers all column.

**Example**: let say we have below dataframe and want to delete the duplicate rows, keeping first occurnace.

df = spark.createDataFrame([

Row(name='Alice', age=5, height=80),

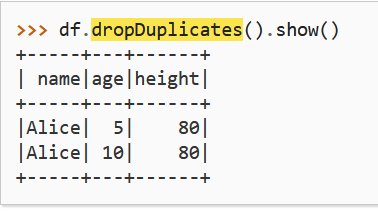
Row(name='Alice', age=5, height=80),

Row(name='Alice', age=10, height=80)

])

**Solution:**

df.dropDuplicates().show()



**##############################**

**# Filter APIs of PySpark DataFrame #**

**##############################**

DataFrame.filter( filterning\_condition, regex=bool, axis=int/str )

This method is used to filter the data from dataframe based on given condition.

* Filtering\_conidtion --- this will be used to filter condition
* Regex=Boolean (optional) --- if want to filter based on regular expression
* Axis = int/str ( option ) ---- axis on which filter will be applied

**Note:**

* use '&' for 'AND', '|' for 'OR' boolean condition
* use column function isin() for multiple search
* use IN operator for SQL Style syntax

dataframe.filter( ( condition1 )| ( condition2 ) | ... ( condition N ) )

dataframe.filter( ( condition1 ) & ( condition2 ) & ... ( condition N ) )

dataframe.filter( ( condition1 ) & ( condition2 ) | ...( condition N ) )

**Note:**

* **In Pyspark comparison is not.**
* **In filter operation only those rows are returned whose comparison result is True**
* **In Pyspark result of comparison can be ( True, False or null/unknown(if any operand) is null)**

<https://spark.apache.org/docs/3.5.4/sql-ref-null-semantics.html>

<https://medium.com/@pcbzmani/pyspark-null-safe-7e79306edca6>

**The following table illustrates the behaviour of comparison operators when one or both operands are NULL**

| **Left Operand** | **Right Operand** | **>** | **>=** | **=** | **<** | **<=** | **<=>**  **Null-safe equality** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| NULL | Any value | NULL | NULL | NULL | NULL | NULL | False |
| Any value | NULL | NULL | NULL | NULL | NULL | NULL | False |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL | True |

**Question:**

Find out all record from dataframe for which ID is less than 3 and name is 'sridevi'

**Solution:**

df.filter( df.ID<3 | (df.NAME == 'sridevi'))

**Question**: Find all completed orders from ord dataframe.

**Solution:**

ord.filter( ( ord.order\_status isin( 'COMPLETED','CLOSED','DELIVERED' ) ) )

**#Question#**

From below dataframe get all rows for which value of **CID column is not '-'**

|  |  |
| --- | --- |
| Input  +----+----------+  | CID|trait\_diff|  +----+----------+  | a| a|  | a| null|  | -| a|  | -| null|  |null| a|  |null| null|  +----+----------+ | Exptd Output  +----+----------+  | CID|trait\_diff|  +----+----------+  | a| a|  | a| null|  |null| a|  |null| null|  +----+----------+ |

**Method1: --- this will not work ( because pyspark comparison is not null safe)**

df.filter(df.CID!='-').show()

**Output**

|  |  |
| --- | --- |
| +---+----------+  |CID|trait\_diff|  +---+----------+  | a| a|  | a| null|  +---+----------+ | **Explanation:**   * **In pyspark comparison operation is not nullSafe** * **If any of the operator is null then result of omparison will be null/unknow** * **In filter operation only those rows are returned whose comparison result is True** |

**Method 2: Correct answer**

df.registerTempTable("my\_temp\_table")

result = spark.sql("SELECT \* FROM my\_temp\_table where !(my\_temp\_table.CID<=>'-') ")

result.show()

**######################**

**# DataFrame Sorting APIs #**

**######################**

For sorting of dataframe we have below two function-

* sort()
* orderBy()

**Note:**

*orderby and sort() works exactly same on dataframe.*

sort() or orderBy() is very costly operation and should be avoided if possible

DataFrame.orderBy(\*cols, \*\*kwargs)

Returns a new DataFrame sorted by the specified column(s).

* **cols** : str, list of Column, optional

list of Column or column names to sort by.

* **ascending** : bool or list, optional. Specifies sort in ascending or descending order each col

DataFrame.sort(\*cols: Union[str, pyspark.sql.column.Column, List[Union[str, pyspark.sql.column.Column]]], \*\*kwargs: Any)

Returns a new DataFrame sorted by the specified column names (cols)

Parameters

**cols :** str, list, or [Column](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column), optional

list of [Column](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) or column names to sort by.

Other Parameters

**ascending :** bool or list, optional

boolean or list of boolean (default True). Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, length of the list must equal length of the *cols*.

**Question:**

Sort the df by age and name in descending order or age and ascending order or name.

**Solution**

df.orderBy(["age", "name"], ascending=[0, 1])

OR

df.orderBy(["age", "name"], ascending=[False, True])

**sortWithinPartitions:**

This is used to sort the data within each partition not the whole data.

DataFrame.sortWithinPartitions(\*cols, \*\*kwargs)

**###################**

**# DataFrame set APIs #**

**###################**

We have below methods for set related operations-

* union() and union All()
* unionByName()
* intersect()
* intersectAll()
* exceptAll()

**DataFrame.union(other)**

Return a new DataFrame containing union of rows in this and another DataFrame.

This does not contains duplicated rows.

This is equivalent to UNION ALL in SQL

If schemas are not the same it returns an error.

DataFrame.unionAll(other)

Return a new DataFrame containing union of rows in this and another DataFrame.

It can contain duplicate of rows.

**Note:**

union and UnionAll() doesn’t creates union by column names. It created union by position

Example:

|  |  |  |
| --- | --- | --- |
| df1 | df2 | df1.union(df2)    We can see that it have pulled 2,3 which are col2 data from df2 in col1 of result because union performs union by position bot by col name. |

**DataFrame.unionByName(other, allowMissingColumns=False)**

Returns a new DataFrame containing union of rows in this and another DataFrame.

*allowMissingCollumns =True*

* Then it takes other columns also in result which are not common and fill value for those column as null

Example: ---- *allowMussingCollumns =False by deafult*

df1 = spark.createDataFrame([[1, 2, 3]], ["col0", "col1", "col2"])

df2 = spark.createDataFrame([[4, 5, 6]], ["col1", "col2", "col0"])

df1.unionByName(df2).show()

+----+----+----+

|col0|col1|col2|

+----+----+----+

| 1| 2| 3|

| 6| 4| 5|

**Note:** We can see here in result that it has *picked columns by name not by position, as col0, col1, col2 are common but different position in each dataframe*.

Example: *allowMussingCollumns = True --- good*

df1 = spark.createDataFrame([[1, 2, 3]], ["**col0**", "col1", "col2"])

df2 = spark.createDataFrame([[4, 5, 6]], ["col1", "col2", "**col3**"])

df1.unionByName(df2, allowMissingColumns=True).show()

+----+----+----+----+

|col0|col1|col2|col3|

+----+----+----+----+

| 1| 2| 3|null|

|null| 4| 5| 6|

+----+----+----+----+

**Note:** Here col0 is missing in df2 and col3 is missing in df1 but present in result, filled by null

**Difference between PySpark uionByName() vs union()**

|  |  |
| --- | --- |
| Union | unionByName |
| * It picks the column by position i.e. column name must be in same order in both dataframe * Applicable if only both datafraem have all columns common in same order   **Check above two example** | * It picks the column name by column name so column names may be in any order in dataframes * All column may not be common and order may not be same   **Check the above sample** |

**df1.intersect(df2)**

Return a new DataFrame containing common rows in both dataframes.

If any rows is more common more than once in both dfs in that case in result it will be only once.

*This is equivalent to* INTERSECT *in SQL*.

**df1.intersectAll(df2)**

Return a new DataFrame containing common rows in both dataframes while preserving duplicates in resultant dataframe.

This is equivalent to INTERSECT ALL in SQL

Example:

Let say we have below dataframe-

df1=spark.sql.createDataFrame(data=[('a',1),('a',1),('b',2)], schema=('col1 string, col2 int'))

df2=spark.sql.createDataFrame(data=[('a',1),('a',1),('c',2)], schema=('col1 string, col2 int'))

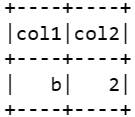
|  |  |
| --- | --- |
| df1.intersectAll(df2)    Intersection with all occurance/duplicates. | df1.intersect(df2)    Intersection with only one occurance |

**df1.exceptAll(df2)**

Rows present in df1 DatFrame but not in df2

This is like minus operation in sql. (df1 – df2)

df1.exceptAll(df2) ---- df1 and df2 are above created dataframe



**######################**

**# JOIN APIs in DataFrame #**

**######################**

For performing join om dataframe we have below methods-

* df1.join(df2,on=column(s), how=type\_of\_join) ---- for performing any type of join
* df1.crossJoin(other) ------- For performsing cross join

df1.join(other, on=None, how=None)

Joins with another DataFrame, using the given join expression.

Parameters are:

**on : str, list or Column, optional**

A string for the join column name, a list of column names, a join expression (Column), or a list of Columns. If on is a string or a list of strings indicating the name of the join column(s), the column(s) must exist on both sides, and this performs an equi-join

**how : str, optional**

default inner. Must be one of: inner, cross, outer, full, fullouter, full\_outer, left, leftouter, left\_outer, right, rightouter, right\_outer, semi, leftsemi, left\_semi, anti, leftanti and left\_anti

**outer:**

* If joining columns are not given the it does cross join
* If joining column are given then does outer join. Check below link for more detail-

[**https://sparkbyexamples.com/pyspark/pyspark-join-explained-with-examples/**](https://sparkbyexamples.com/pyspark/pyspark-join-explained-with-examples/)

**See difference in below two output (empDF,deptDF – above link )**

1. **empDF.join(deptDF,empDF.emp\_dept\_id = = deptDF.dept\_id,"outer").show()**
2. **empDF.join(deptDF,how="outer")()**

Example1: --- specify the column names from both dataframe on which join will be perform

df.join(df2, df.name == df2.name, 'outer')

Example 2: --- Specify the the common column names once

df.join(df2, on='name', 'outer')

Example 3: Joining on more than one column

cond = [df.name == df3.name, df.age == df3.age]

df.join(df3, on=column\_name\_for\_join, 'outer')

**Note:**

We can specify the join in below way if joining on multiple columns.

df1.join( df2, **(df1.col1=df2.col1) & ( df1.col2 == df2.col2 )** )

Example 4:

df.join(df4, ['name', 'age'])

**# Joingin more than two dataframe #**

If we have more than 2 dataframe then we can join using chaining concept.

Result of one join will be used to another join.

df1.join( df2, on=joining\_col, how=type\_of\_join).join( df3, on=condition, how=type)

df.count()

Returns the number of rows in this DataFrame.



**##############**

**# GroupBy APIs #**

**##############**

**DataFrame.groupBy(\*cols)**

Groups the DataFrame using the specified columns, so we can run aggregation on them.

If applied on any dataframe it return GroupData object.

**cols : list, str or Column**

columns to group by. Each element should be a column name (string) or an expression

**Note**:

For more on group by check for GROUPING API (search for GROUPING API in this file)

**We can apply below aggregate function on groupBy object-**

* mean()
* count() --- it does not take any column name as input
* countDistinct(\*col)
* min(\*cols)
* max(\*cols)
* sum(\*cols)
* aff(\*cols) ---- for multiple aggregation at time
* pivot()
* apply()
* avg() ---- this is used to apply multiple aggregate function at a time.

**\*cols** – name of column (one or more) on which we can to apply the aggregation.

**Note:**

* We can group on one or more column names and can apply aggrerate function on one/more columns.
* All the above aggregation functions are applied on a column
* Aggregation function can be applied on same column on which group by is applied.

e.g - df.**groupBy('A').agg(count(col("A")))**.show() )

* After aggregate function application only those column will be present on which groupBy and aggretation is applied.

Let say we have below dataframe which we are going to use for groupby and aggregate function leanring-

|  |  |
| --- | --- |
| simpleData = [("James","Sales","NY",90000,34,10000),  ("Michael","Sales","NY",86000,56,20000),  ("Robert","Sales","CA",81000,30,23000),  ("Maria","Finance","CA",90000,24,23000),  ("Raman","Finance","CA",99000,40,24000),  ("Scott","Finance","NY",83000,36,19000),  ("Jen","Finance","NY",79000,53,15000),  ("Jeff","Marketing","CA",80000,25,18000),  ("Kumar","Marketing","NY",91000,50,21000)  ]  schema = ["employee\_name","department","state","salary","age","bonus"]  df = spark.createDataFrame(data=simpleData, schema = schema)  print(df.show(2)) |  |

**Question:**

Get the sum of salary for each detapartment.

**Solution:**

|  |  |
| --- | --- |
| df.groupBy(df.department).sum('salary'). show()  OR  df.groupBy("department").agg(sum("salary").alias('sum\_of\_salary'))  **Note:**  As from result we can see only department and salary/sum(salary) is present in output because in result only groupBy and aggregated column will be present. |  |

**Question:**

Get the min salary and bonus for each state in each department.

**Solution:**

df.groupBy(df.department,df.state).min('salary','bonus').show()

+----------+-----+-----------+----------+

|department|state|min(salary)|min(bonus)|

+----------+-----+-----------+----------+

| Finance| NY| 79000| 15000|

| Marketing| NY| 91000| 21000|

| Sales| CA| 81000| 23000|

| Marketing| CA| 80000| 18000|

| Finance| CA| 90000| 23000|

| Sales| NY| 86000| 10000|

+----------+-----+-----------+----------+

**df.agg()**

this is used to apply multiple aggregate function on groupBy dataframe object.

df.groupBy(grouping\_cols).agg( aggregate\_func1, aggregate\_func2 . . . aggregate\_fun<n> )

**Question:**

For above dataframe get the min, max and average of salary for each deparemtn.

**Solution:**

from pyspark.sql.functions import \*

df.groupBy(df.department).agg(min("salary"),max("salary"),avg("salary")).show()

+----------+-----------+-----------+-----------------+

|department|min(salary)|max(salary)| avg(salary)|

+----------+-----------+-----------+-----------------+

| Sales| 81000| 90000|85666.66666666667|

| Finance| 79000| 99000| 87750.0|

| Marketing| 80000| 91000| 85500.0|

+----------+-----------+-----------+-----------------+

**# where with grouping and aggregation #**

* We should apply where before applying the groupBy function on a column on which grouping or aggregation is not applied. (because only groupBy and aggregate column are present in output.)
* We can apply where after the aggregation and groupBy if those columns are present in result.

**Question:**

Find the min, max and average of salary for each department for NewYork state.

**Solution:**

from pyspark.sql.functions import \*

df.where(df.state=='NY').groupBy(df.department).agg(min("salary"),max("salary"),avg("salary")).show()

+----------+-----------+-----------+-----------+

|department|min(salary)|max(salary)|avg(salary)|

+----------+-----------+-----------+-----------+

| Sales| 86000| 90000| 88000.0|

| Finance| 79000| 83000| 81000.0|

| Marketing| 91000| 91000| 91000.0|

+----------+-----------+-----------+-----------+

**Question:**

Find the min, max and average of salary for each department where minimum salary is more than 20000.

**Solution:**

df.groupBy(df.department).agg(min("salary").alias("min\_salary"),max("salary"),avg("salary"))\

.where( **col("min\_salary")** >20000).show()

**##################################**

**# Window and Window function usages #**

**##################################**

In PySpark we have below two window classes-

* Window
* WindowSpec

On created window we use window function ( rank, row\_number, dense\_rank etc) to doing statical calculation

We have below methods of window class-

|  |  |
| --- | --- |
| [Window.orderBy](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Window.orderBy.html#pyspark.sql.Window.orderBy)(\*cols) | Creates a WindowSpec with the ordering defined. |
| [Window.partitionBy](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Window.partitionBy.html#pyspark.sql.Window.partitionBy)(\*cols) | Creates a WindowSpec with the partitioning defined. |
| [Window.rangeBetween](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Window.rangeBetween.html#pyspark.sql.Window.rangeBetween)(start, end) | Creates a WindowSpec with the frame boundaries defined, from *start* (inclusive) to *end* (inclusive). |
| [Window.rowsBetween](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Window.rowsBetween.html#pyspark.sql.Window.rowsBetween)(start, end)  Can be used for ffill or bfill  Operation.  1.end>start --- ffill operation  e.g -- Start=-inf and end=0  2. end<start --- bfill operation  e.g -- end=-inf and start=0  **Search– forward\_fill and backward\_fill concept in doc** | Creates a WindowSpec with the frame boundaries defined, from *start* (inclusive) to *end* (inclusive).  window = Window.rowsBetween(float('-inf'),0) --- forward frame  window = Window.rowsBetween(0, float('-inf'),) --- backward frame |

|  |  |
| --- | --- |
| [WindowSpec.orderBy](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.WindowSpec.orderBy.html#pyspark.sql.WindowSpec.orderBy)(\*cols) | Defines the ordering columns in a WindowSpec. |
| [WindowSpec.partitionBy](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.WindowSpec.partitionBy.html#pyspark.sql.WindowSpec.partitionBy)(\*cols) | Defines the partitioning columns in a WindowSpec. |
| [WindowSpec.rangeBetween](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.WindowSpec.rangeBetween.html#pyspark.sql.WindowSpec.rangeBetween)(start, end) | Defines the frame boundaries, from *start* (inclusive) to *end* (inclusive). |
| [WindowSpec.rowsBetween](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.WindowSpec.rowsBetween.html#pyspark.sql.WindowSpec.rowsBetween)(start, end) | Defines the frame boundaries, from *start* (inclusive) to *end* (inclusive). |

**Creating window:**

Win\_spec=Window\_class.window\_class\_func

* Window\_class= Window or WindowSpec
* Window\_class\_func= any function of window or WindowSpec class

**Using window on dataframe**

**applicable\_function().over(wind\_spec\_object)**

**applicable\_function** --- we can apply any function from **ranking, analytical, aggregate from below**.

* **Ranking Window function** – rank(), row\_number(), dense\_rank() – Same as SQL
* **Analytical Window function** – lag(), lead () – Same as LAG and LEAD of SQL
* **Aggregate Window function** – avg(), sum(), min(), max(), count(), first(), last()

**Window\_function** ===> rank(), row\_number(), dense\_ranke(), lead(), lag(), avg() etc **(check below)**

**Wind\_spec\_obj** ===> window object created using Window class and it’s method **( check above)**

Note:

* While creating window atleast one of method orderBy must be used, partitionBy is optional.

( Check example --- "Example 36" )

* If partitionBy is not given or given but no column name specified in partitionBy on window then by default it takes whole column for partitioning created the result. *Let say in one dataframe we have col1, col2 . . . coln then below two statement are same:-*
  + **Window.partitionBy(co1,col2,. . . coln).orderBy(<cols\_to\_order\_comma\_sperated>)**
  + **Window.orderBy(<cols\_to\_order\_comma\_sperated>)**
* Default window/frame is - Window.UnboundedPreceding to Window.CurrentRow
* **Check example --- #Example**# **window /frame size and direction in window**

**Note: ---- from stackoverflow , Check question "#Question# --- aggregation on window" --- Impt**

* When we apply aggregation on window then it applies that aggregation function on current data and previous data. Check question from below link for more clear picture-

<https://stackoverflow.com/questions/79084670/create-column-of-current-balance-in-pyspark/79084677#79084677>

**Note:**

By default orderBy takes ascending order, to control it we can use in below way-

* orderBy( df.col\_name.desc() ) --- For descending order
* orderBy( df.col\_name.asc() ) --- For ascending order

**Question:**

Create/assign the rank for each Salary within each department.

**Solution:**

Step1: --- Create a window on dataframe ( using partitionBy on department and orderBy on salary ).

Spec=Window.partitionBy( dept )

Step2: --- Now apply the window function

from pyspark.sql.functions import \*

from pyspark.sql.window import Window

#create the window or spec

spec=Window.partitionBy("department").orderBy("salary")

#Now apply the window\_function

df.select(df.department,df.salary).withColumn("ranking",dense\_rank().over(spec) ).show()

+----------+------+-------+

|department|salary|ranking|

+----------+------+-------+

| Finance| 79000| **1**|

| Finance| 83000| **2**|

| Finance| 90000| **3|**

| Finance| 99000| **4**|

| Marketing| 80000| **1**|

| Marketing| 91000| **2**|

| Sales| 81000| **1**|

| Sales| 86000| **2**|

| Sales| 90000| **3**|

+----------+------+-------+

**Question**

Get the employee data from employee table of hr database and assign a rank for each employee of each department in descending order of salary i.e highest salary lowest rank (1).

**Solution:**

from pyspark.sql.functions import \*

from pyspark.sql.window import Window

#Now read the data from hr database

df=spark.read.format('jdbc').options(url="jdbc:mysql://localhost:3306/hr",dbtable = "(select \* from employees) query\_alias\_name",user="root",password="admin").load()

#now partiion by each department id and provide the rank bases on salary

spec=Window.partitionBy('DEPARTMENT\_ID').orderBy(df.SALARY.desc())

df.withColumn('rank',dense\_rank().over(spec)).select(df.FIRST\_NAME,df.LAST\_NAME,df.SALARY,df.DEPARTMENT\_ID,'rank').show(4)

**#Question #** -- Search by Question 38 , in this doc.

Find the nth (n= any integer number) largest element from given dataframe.

|  |  |
| --- | --- |
| +---+-------+  | id|column1|  +---+-------+  | 1| 5|  | 2| 8|  | 3| 12|  | 4| 1|  | 5| 15|  | 6| 7|  +---+-------+ | 3rd largest element will be – 8  1rst largest element will be – 15 |

**Analytical Window Function**

1. lag(col, offset=1, default=None)
2. lead(col, offset=1, default=None)

* col--- column name or column object
* offset --- int, number of row to extend
* default ---- None

lag function example:

from pyspark.sql.functions import \*

from pyspark.sql.window import Window

#create the window or spec

spec=Window.partitionBy("department").orderBy("salary")

#Now fo lag opration

df.select(df.department, df.salary).withColumn("lag\_salary\_value",lag(df.salary,1,0).over(spec)).show()

+----------+------+----------------+

|department|salary|lag\_salary\_value|

+----------+------+----------------+

| Finance| 79000| 0|

| Finance| 83000| 79000|

| Finance| 90000| 83000|

| Finance| 99000| 90000|

| Marketing| 80000| 0|

| Marketing| 91000| 80000|

| Sales| 81000| 0|

| Sales| 86000| 81000|

| Sales| 90000| 86000|

+----------+------+----------------+

**#Question** # Difference between rowsBetween() and rangeBetween()

<https://www.youtube.com/watch?v=4eEIs9gtKjw>

**Column.alias( \*alias\_name, \*\*kwargs)**

This is used for aliasing a column name.

This is used generally when we are doing some operation that creates a new column.

**Example:**

df=spark.read.csv('D:\\PySpark\\Prac1\\online\_retail.csv',header=True)

df2=df.groupBy('Country').agg(F.count('Country').**alias('buy\_count')** )

print(df2.show(2))

**Note:**

If column name is already existing then we cannot alias function to change it’s name, we should use *withRename() function for column renaming*

This is used for aliasing a column name.

**Aggregate function in pyspark**

In pyspark we have below aggregate function and we can use then with window function or even without window function for aggregation.

<https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/>

* [approx\_count\_distinct](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#approx-count-distinct)
* [avg](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#avg)
* [collect\_list](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#collect-list)
* [collect\_set](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#collect-set)
* [countDistinct](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#countDistinct)
* [count](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#count)
* [grouping](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#grouping)
* [first](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#first)
* [last](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#last)
* [kurtosis](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#kurtosis)
* [max](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#max)
* [min](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#min)
* [mean](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#mean)
* [skewness](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#skewness)
* [stddev](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#stddev)
* [stddev\_samp](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#stddev)
* [stddev\_pop](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#stddev)
* [sum](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#sum)
* [sumDistinct](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#sumDistinct)
* [variance](https://sparkbyexamples.com/pyspark/pyspark-aggregate-functions/#variance), var\_samp, var\_pop

**Note:**

* NULL values are ignored from processing by all the aggregate functions

**###############**

**# Sampling APIs #**

**###############**

For sampling of dataframe in spark we have below two functions-

1. df.sample() ---- Same as RDD sample()
2. df.sampleBy() ---- Same as RDD sampleBy()

**dataFrame.sample(withReplacement=None, fraction=None, seed=None)**

Return a sampled subset of dataframe.

* withReplacement : bool, optional, default=False

same result can be produced more than once

* fraction : float, optiona , values could be in [0.0 1]

fraction of total record to consider for taking sample

* seed : int, optional

seed for sampling ( default a randon seed)

**DataFrame.sampleBy(col, fractions, seed=None)**

Returns a stratified sample without replacement based on the fraction given on each stratum.

* col: column or str

column that defined strata

* fraction: dict

sampling fraction for each stratum

* seed: int, optinal

**DataFrame.select(\*cols)**

Projects a set of expressions and returns a new DataFrame. We can use it in below difference ways-

1. df.select( df.col\_name1, df.col\_name2 . . . )
2. df.select ( 'col\_name1','col\_name2',. . . . )
3. df.col\_name1+y,'col\_name2'+x,. . . .

Example:

ord=spark.read.load("D:\\Udemy\_PySpark\\RetailDB+SalesData\\RetailDBSalesData\\Orders\\order",sep=',',format='csv',schema=('order\_id int,order\_date timestamp,customer\_id int,order\_datatus string'))

print(ord.show())

**Selecting the required columns from dataframe.**

cols: str, Column, or list

column names (string) or expressions (Column).

If one of the column names is ‘\*’, that column is expanded to include all columns in the current DataFrame.

**Question:**

Find out all unique country from csv file.

**Solution:**

df = spark.read.csv('D:\\PySpark\\Prac1\online\_retail.csv',header=True)

#Get the uniq contry

print(df.select('Country').drop\_duplicates().show())

Question:

Get the count of distinct customerId for each country

Solution:

import pyspark.sql.functions as F

df=spark.read.csv('D:\\PySpark\\Prac1\\online\_retail.csv',header=True)

print('completed reading of file')

print(df.show(2))

#print(df.groupBy("Country").count().show(2))

gr = df.groupBy("Country").agg(F.countDistinct("CustomerID"))

print(gr.show())

**DataFrame.withColumn(colName, col\_expr)**

This *method is used to create/rename column by using column name from same dataframe*.

The col\_expr must be an expression over same DataFrame; attempting to add a column from some other DataFrame will raise an error.

*If colName is already present in dataframe then value will be replaced by new value*

*If colName is not present then a new column will be inserted in dataframe with new value.*

Example:

Add 2 in age column of a dataframe

Solution:

df.withColumn(‘age’, df.age + 2) --- will add 2 in age column value and will replace

df.withColumn('age2', df.age + 2) --- will add new column by name age2

DataFrame.withColumnRenamed(old\_col\_name, new\_col\_name)

Returns a new DataFrame by renaming 'old\_col\_name' to 'new\_col\_name'

If old\_col\_name is not present then no operation will be performed.

**Example:**

Rename the country column to 'Rashtra'.

**Solution**

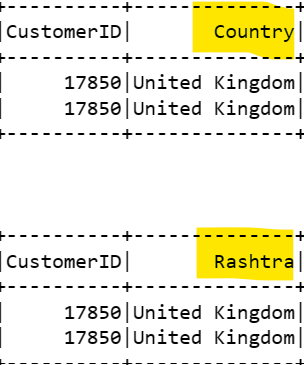
df=spark.read.csv('D:\\PySpark\\Prac1\\online\_retail.csv',header=True)

print('completed reading of file')

print(df.show(2))

df1=df.withColumnRenamed('Country','Rashtra')

print(df1.show(2))



################################################################################

# DataFrame Built-in function #

################################################################################

We can categorized dataframe built-in function in below catageory-

1. New Column
2. Encryption
3. String
4. RegExp
5. Date
6. Null
7. Collection
8. Na
9. Math and Statistics
10. Explode and Flatten
11. Formatting
12. Json
13. Other

**################**

**# String Function #**

**################**

We have below string function which are commonly used-

All these methods are in pyspark.sql.function module.

1. **concat(\*cols)**

Concatenates multiple input columns together into a single column. The function works with strings, binary and compatible array columns.

1. **pyspark\_col.contains(str\_or\_col\_object)**

Checks if column contains str in data. Returns a boolean Column based on a string match.

1. **pyspark\_col.startswith(str\_or\_col\_object)-- pyspark.sql.function**

Check of column data startswith given str and returns boolean.

1. **pyspark\_col.endswith(str\_or\_col\_object)**

Used to check if the column data ends with given string str and returns boolean.

1. **initcap(pyspark\_col\_or\_string)**

Used to convert the first letter of each column data into uppercase.

1. **upper(pyspark\_col\_or\_string)**

It converts a string column name ( pyspark\_col ) to uppercase.

1. **lower(pyspark\_col\_or\_string)**

It converts a string column name ( pyspark\_col ) to lower case.

1. **substring(pyspark\_col\_name, start, end ) OR col\_obj.substr(startPos, length)**

This column is used to extract the substring from given column name ( pyspark\_col\_name )

**Question:**

From ord dataframe extract the year value from order\_data column and assign to new column.

**Solution:**

from pyspark.sql.functions import \*

#extract the year value and assign to year column

ord=ord.withColumn('year',substring(ord.order\_date,0,4))

print(ord.show(2))

1. **pyspark\_col\_obj.substr(startPos, length)**

Return a column which is substring of the column.

1. **length(col\_object\_or\_string\_name)**

this function is used to calculate the lenght of each data column into subtring.

1. **lpad(col\_object\_or\_string,, len, pad)**

Left-pad the string column to width len with pad.

1. **rpad(col\_obj\_or\_string,, len, pad)**

Right-pad the string column to width len with pad.

1. **trim(col\_object\_or\_string)**

Trim the spaces from both ends for the specified string column.

1. **ltrim(col\_object\_or\_string,)**

Trim the spaces from left ends for the specified string column.

1. **rtrim(col\_object\_or\_string,)**

Trim the spaces from right ends for the specified string column.

1. **split(col\_object\_or\_string, pattern, limit=- 1)**

Splits str around matches of the given pattern and return in list.

**Question:**

From ord dataframe extract the year value from order\_data column and assign to new column.

**Solution:**

from pyspark.sql.functions import \*

#split agains the '-' and pick the first value from splitted data

ord=ord.withColumn('year',split(ord.order\_date,'-')[0])

print(ord.show(2))

########################

# PySpark Reg Exp function #

#######################

1. **regexp\_extract(str\_or\_col\_object, pattern, idx)**

Extract a specific group matched by a Java regex, from the specified string column.

Returns the empty string is specified pattern is not found.

Pattern is given in single/double quote

**Idx ---** this represents the group number which wanted to extract from matched groups.

**Min(idx)=1**

**Example**: Create a dataframe and extract all numerical data.

**Solution**:

from pyspark.sql.functions import \*

df=spark.createDataFrame(["hello","132hekdl233","13jdsnnj32kjds"], "string").toDF("age")

print(df.show())

#now filter out all number

print(df.select('age',regexp\_extract(df.age, '(\d+)', 1)).show())

1. **regexp\_replace(col\_object\_or\_col\_name\_or\_string,, pattern, replacement)**

Replace all substrings of the specified pattern by 'replacement' of column object.

Pattern is given in single/double quote

**Example**: Relace the all number from above dataframe by 'Z'.

**Solution:**

from pyspark.sql.functions import \*

df=spark.createDataFrame(["hello","132hekdl233","13jdsnnj32kjds"], "string").toDF("age")

print(df.show())

#now replace the all number by Z

print(df.select('age',regexp\_replace(df.age, '\d', 'Z')).show())

1. **col\_object.rlike(pattern\_or\_string)**

This function is used to check if pattern is present in column or not.

It takes a literal regex expression string as a parameter and returns a boolean column based on a regex match.

#####################

# PySpark date function #

#####################

We have below methods in pyspark to work with date or datetime.

All of these are available in pyspark.sql.functions

* **current\_date()** ---- Gives the current date
* **curernt\_timestamp()**
* **next\_day(date, dayOfWeek)**

Retuen the first date of dayOfWeek which is after the 'date'.

dayOfWeek = "Mon","Tue",. . . "Sun"

* **last\_day(date)**

Returns the last day of the month which the given date belong to.

* **dayofweek(col)**

retunrs the day number of a week for which given day belongs to

* **dayofmonth(col\_or\_date)**

Return the day number of given date for which date belong to,

* **dayofyear(date\_or\_col)**

Return the day of of year for given date

* **weekofyear(col\_or\_date)**

Return the number number of the year for which given date nbelongs to.

* second(date\_of\_col)

Extracts the second of a given date as integer

* minute(date\_of\_col)

Extracts the minute of a given date as integer

* hour(date\_of\_col)

Extracts the hour of a given date as integer

* month(date\_of\_col)

Extracts the month of a given date as integer

* quarter(date\_of\_col)

Extracts the quarter of a given date as integer

* year(date\_of\_col)

Extracts the year of a given date as integer

* month\_between(date1, date2)

Difference of two date in month as float

* date\_add(start\_date\_or\_col, days)

Add the given number of 'days' into start date and return it

* date\_sub(date\_or\_col, days)

Substract the given number of 'days' from date

* add\_months( start, months)

add the given number of months into start date

* datediff(end, start)

Retuen the number of days from start to end

* **date\_format(date, format)**

Converts a date/timestamp to string format ( like python datetime.strftime(format) ).

* size(col\_name)

Returns the lenght of the array or map stored in the column col\_name.

col\_name must be array of list type

Size if -1 for null elements.

* element\_at(col\_name, extraction)

Returns elemet of array at given inex in extraction if col\_name is array.

Return value for the given key in extraction if col is map.

* slice(col\_name , start, lenght )

this is used to slice the data from column starting from start of length number of character

**################################**

**# Creating new column in dataframe #**

**################################**

We can create column in dataframe based on many condition.

**Add New Column with Constant Value**

To *add a new column to DataFrame use lit()* function by importing from pyspark.sql.functions import lit , lit() function takes a constant value you wanted to add and returns a Column type, if you wanted to add a NULL/None use lit(None)

**df.withColumn("new\_col\_name", lit(contant\_value) )**

**Add Column Based on Another Column of same DataFrame**

df.withColumn("bonus\_amount", <calculating\_from\_existing\_col>)

df.withColumn(‘hike’, df.salary\*.20) ---- To add a new column hike whose value will be 20% of salary

**Add Column Value Based on Condition**

df.withColumn( 'new\_col\_name', condtion\_to\_assign\_value' )

Example:

df.withColumn("grade", when((df.salary < 4000), lit("A")).when((df.salary >= 4000) & (df.salary <= 5000), lit("B")).otherwise(lit("C"))

**#Getting dataframe when column values are in list#**

df.filter( df.col\_name.isin( values\_list ) )

**# Aggregation of dataframe #**

For aggregating we can aggregate dataframe in two ways-

1. Normal aggregation
2. Aggregation on grouped data

**Normal aggregation**

**DataFrame.agg(\*exprs: Union[pyspark.sql.column.Column, Dict[str, str]])**

Aggregate on the entire DataFrame without groups (shorthand for df.groupBy().agg())

We can apply agg function on more than one column. (like pandas agg)

**We can apply agg function on same column on which grouping is done.**

Parameter:

exprs: Column or dict of key and value strings

Columns or expressions to aggregate DataFrame by.

df.agg ( {col\_name : function\_to\_apply\_on\_col }) **-------- Syntax 1**

df.agg( aggregation\_function1, . . ,aggregation\_functionN) **-------- Syntax 1**

this is basically column name, and function to apply on that column

OR

it can be a aggerated function appied on a column

**Example1: -------- Syntax1**

Find the maximum age of column data from df dataframe.

**Solution:**

df.agg( {"age": "max"} ).show()

**Example2: -------- Syntax2**

Find the minimum value of age.

**Solution:**

df.agg( min(df.age), max(df.age) ).show()

**Aggregation on grouped data**

Aggregation on grouped data is exactly same as normal aggregation except here we apply agg function on aggregated data.

**Grouped\_data.agg(aggregation\_function)**

**Example:**

iris1\_df1.groupBy('Species').agg({'Sepal\_Length':'mean'}).show()

**###################################**

**#ROW in DataFrame ---- pyspark.sql.row #**

**###################################**

In PySpark Row class is available by importing pyspark.sql.Row which is represented as a record/row in DataFrame, one can create a Row object by using named arguments, or create a custom Row like class.

**Creating Row object:**

We can create Row object is many way-

1. Creating Row object with label name
2. Creating Row object without label
3. Creating Row object by Custom Row class

**Creating row object with label names:**

row= Row( label1 = value, label2 = value2 . . . labeln = valuen)

**Note:**

* When we create Dataframe using the row object then **it treats the label as columns of dataframe.**
* If creating dataframe using row object with label but still we can pass columns name with schema variable.

**Creating Row object without lavel**

row= Row(value, value2 , . . . valuen)

**Creating Row object from Custom Row class**

My\_Class = Row( label1, label2, . . . . labeln) --- Custom Row class created

p1= My\_Class (value1, value2, value3, . . . valuen) --- Row object from custom Row class

p2= My\_Class (value1, value2, value3, . . . valuen) --- Row object from custom Row class

**##############################**

**# Accessing data from Row object #**

**##############################**

The fields in it can be accessed in below way-

1. row\_object.label\_name --- gives the value of label\_name in row\_object
2. row\_object[label\_name]) -- gives the value of label\_name in row\_object
3. lable\_name in row\_object --- Checks if label\_name is in row\_object

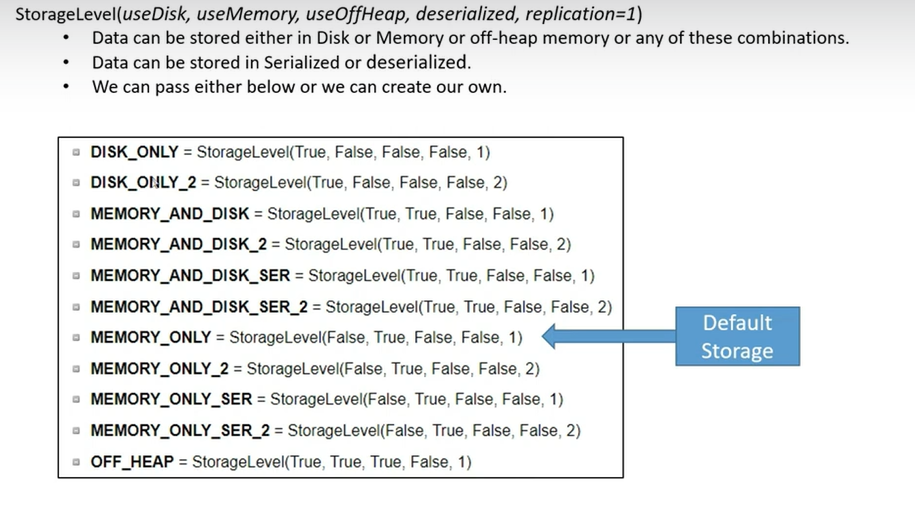
**Methods of Row Object:**

|  |  |
| --- | --- |
| [asDict](https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.Row.asDict.html#pyspark.sql.Row.asDict)([recursive]) | Return as a lable\_name and values as dict |
| count(value, /) | Return number of occurrences of value. |
| index(value[, start, stop]) | Return first index of value. |

**##############################**

**# DataFrame Organization of data #**

**##############################**



**################################################################################**

**# DataTypes #**

**################################################################################**

We have below data type in pyspark/spark. These are usually used in defining the schema while creating rdd or dataframe.

|  |  |
| --- | --- |
| [ArrayType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.ArrayType.html#pyspark.sql.types.ArrayType)(elementType[, containsNull]) | Array data type. |
| [BinaryType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.BinaryType.html#pyspark.sql.types.BinaryType) | Binary (byte array) data type. |
| [BooleanType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.BooleanType.html#pyspark.sql.types.BooleanType) | Boolean data type. |
| [ByteType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.ByteType.html#pyspark.sql.types.ByteType) | Byte data type, i.e. |
| [DataType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.DataType.html#pyspark.sql.types.DataType) | Base class for data types. |
| [DateType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.DateType.html#pyspark.sql.types.DateType) | Date (datetime.date) data type. |
| [DecimalType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.DecimalType.html#pyspark.sql.types.DecimalType)([precision, scale]) | Decimal (decimal.Decimal) data type. |
| [DoubleType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.DoubleType.html#pyspark.sql.types.DoubleType) (8-byte) | Double data type, representing double precision floats. |
| [FloatType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.FloatType.html#pyspark.sql.types.FloatType) (4-byte) | Float data type, representing single precision floats. |
| [IntegerType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.IntegerType.html#pyspark.sql.types.IntegerType) (4-byte ) | Int data type, i.e. |
| [LongType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.LongType.html#pyspark.sql.types.LongType) | Long data type, i.e. |
| [MapType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.MapType.html#pyspark.sql.types.MapType)(keyType, valueType[, valueContainsNull]) | Map data type. |
| [NullType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.NullType.html#pyspark.sql.types.NullType) | Null type. |
| [ShortType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.ShortType.html#pyspark.sql.types.ShortType) | Short data type, i.e. |
| [StringType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.StringType.html#pyspark.sql.types.StringType) | String data type. |
| [CharType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.CharType.html#pyspark.sql.types.CharType)(length) | Char data type |
| [VarcharType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.VarcharType.html#pyspark.sql.types.VarcharType)(length) | Varchar data type |
| [StructField](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.StructField.html#pyspark.sql.types.StructField)(name, dataType[, nullable, metadata]) | A field in [StructType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.StructType.html#pyspark.sql.types.StructType). |
| [StructType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.StructType.html#pyspark.sql.types.StructType)([fields]) | Struct type, consisting of a list of [StructField](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.StructField.html#pyspark.sql.types.StructField). |
| [TimestampType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.TimestampType.html#pyspark.sql.types.TimestampType) | Timestamp (datetime.datetime) data type. |
| [TimestampNTZType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.TimestampNTZType.html#pyspark.sql.types.TimestampNTZType) | Timestamp (datetime.datetime) data type without timezone information. |
|  |  |

MapType(**keyType**: pyspark.sql.types.DataType, **valueType**: pyspark.sql.types.DataType, **valueContainsNull**: bool = True)

This is usually used when we create struct/schema for semi-structure data.

This is basically for case when column contains python dictionary data or map in java.

Parameter:

* Keytype , Value type : These are data type of key and valye and should be from pyspark.sql.types.
* valueContainsNull : this is Boolean value to check if column contains boolean value.

**Example**:

Create the dataframe having columns – name (string value) and properties (dictionary type)

+----------+-----------------------------+

|Name |properties |

+----------+-----------------------------+

|James |[eye -> brown, hair -> black]|

|Michael |[eye ->Red, hair -> null] |

|Washington|[eye ->Brown] |

+----------+-----------------------------+

**Solution**:

from pyspark.sql.types import StructField, StructType, StringType, MapType

#define the schema

schema = StructType([

    StructField('name', StringType(), True),

    StructField('properties', MapType(StringType(),StringType()),True)

])

**Map vs Struct**

Map has key-value pairs, and struct also can be visualized to have some sort of key-value pairs. So, how do we know where to use Map and where to use Struct.

* If there are some keys for a pyspark column dataframe which need to be defined for every record, then we can make **use of Struct**.
* If there needs to be key-value pairs for pyspark column dataframe, where the keys are dynamic, then we can make **use of Map**.

**StructType**: When a column of pyspark have fixed sub-column(name) and fixed data type of sub-column value for each row.

**MapType**: When column of pyspark don't have any sub-column(name) then use MapType.

|  |  |
| --- | --- |
| **MapType**    Utilities column have sub-column but are dynamic. | **StructType**    *Name column have 3 sub-column (FirstName, MiddleName, LastName) and will have some value(including null) for each row of Name column*. |

**Example:**

Let say we have requirement like , we will have name/object and properties associated with them.

[( Rohan, { eye : brown, hair: black} ), ( Shyam, { eye : black, hair: black} )]

Create the structype for such case.

**Solution**:

from pyspark.sql.types import StructField, StructType, StringType, MapType

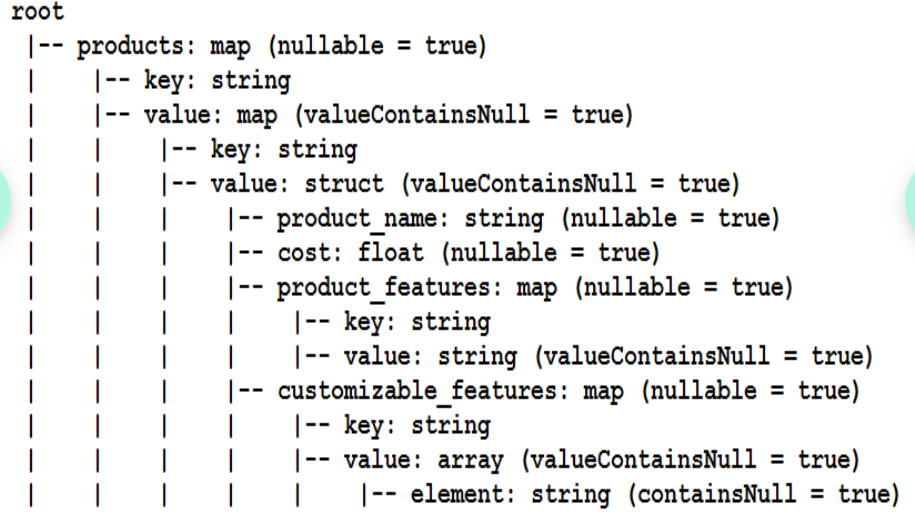
schema = StructType([

StructField('name', StringType() ),

StructField('properties', MapType(StringType(),StringType()),True)

])

Example 2: We have below schema requirement, create sctruct details-



Solution:

from pyspark.sql.types import StructType, StructField, MapType, ArrayType, FloatType, StringType

schema1 = StructType([

    StructField(

        'products',

        MapType(

            StringType(),

            MapType(

                StringType(),

                StructType([

                    StructField('product\_name',StringType()),

                    StructField('cost',FloatType()),

                    StructField(

                        'product\_features',

                        MapType(StringType(),StringType())

                    ),

                    StructField(

                        'customizable\_features',

                        MapType(

                            StringType(),

                            ArrayType(StringType())

                        )

                    )

                ])

            )

        )

    )

])

**Accessing Struct Element:**

Struct elements can be accessed **using dot operator**.

**Example**:

retailer1.selectExpr('products.watch').show()

**Accessing Map Elements**

Map elements can be accessed **using dot operator**.

**Example:**

retailer1.selectExpr('products.watch.casio').show()

**Accessing Array Elements**

Array elements can be accessed **using the index.**

**Example:**

retailer1.selectExpr('products.watch.casio.customizable\_features.back\_light\_color\_options[0] as new\_col').show()

**Note**:

Instead of selectExpr we can use select, just replace selectExpr by select, no other changes.

**#creating table from DataFrame#**

We can create table from dataframe. Creating table allows us to work using sql query on created table.

We can create table using any of below methods.



All above method creates table just difference is life or table.

**Question:**

Create a temporary table for iris.csv file.

**Solution**:

iris=spark.read.csv('D:\\PySpark\\Prac1\\iris\\iris.csv',header=True)

print(iris.show(2, truncate=False))

iris.createOrReplaceTempView('temp\_table')

**#Accessing data from table#**

For accessing data from a table we need to give the sql query in spark\_session.sql( query ) function.

This methods returns the result of query as dataframe.

iris=spark.read.csv('D:\\PySpark\\Prac1\\iris\\iris.csv',header=True)

print(iris.show(2, truncate=False))

iris.createOrReplaceTempView('temp\_table')

#df.createTempView(name: str)

res=spark.sql("select \* from temp\_table")

print(type(res))#<class 'pyspark.sql.dataframe.DataFrame'>

print(res.show(2))

**#converting table into dataframe#**

We can convert a table into dataframe using table( table\_name ) function from spark session.

spark\_session.table( table\_name ) --- Creates dataframe from table\_name

**Example:**

Covert the above create table from iris.csv file into dataframe.

**Solution:**

df= spark\_session.table(‘temp\_table’ ) ---- it will convert temp\_table into dataframe.

**################################################################################**

**# pyspark.sql.column Column APIs #**

**################################################################################**

**pyspark.sql.Column class** provides several functions to work with DataFrame to manipulate the Column values, evaluate the boolean expression to filter rows, retrieve a value or part of a value from a DataFrame column, and to work with list, map & struct columns

PySpark Column class represents a single Column in a DataFrame.

We have below methods for column API **and work on applies on each element for called column**..

We will see few most used in details-

|  |  |
| --- | --- |
| [Column.alias](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.alias.html#pyspark.sql.Column.alias)(\*alias, \*\*kwargs) | Returns this column aliased with a new name or names (in the case of expressions that return more than one column, such as explode). |
| [Column.asc](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.asc.html#pyspark.sql.Column.asc)() | Returns a sort expression based on ascending order of the column. |
| [Column.asc\_nulls\_first](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.asc_nulls_first.html#pyspark.sql.Column.asc_nulls_first)() | Returns a sort expression based on ascending order of the column, and null values return before non-null values. |
| [Column.asc\_nulls\_last](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.asc_nulls_last.html#pyspark.sql.Column.asc_nulls_last)() | Returns a sort expression based on ascending order of the column, and null values appear after non-null values. |
| [Column.astype](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.astype.html#pyspark.sql.Column.astype)(dataType1,dtyp2..dtypen) | astype() is an alias for cast(). |
| [Column.between](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.between.html#pyspark.sql.Column.between)(lowerBound, upperBound) | Returns True if the current column is between the lower bound and upper bound, inclusive else Flase |
| [Column.bitwiseAND](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.bitwiseAND.html#pyspark.sql.Column.bitwiseAND)(other) | Compute bitwise AND of this expression with another expression. |
| [Column.bitwiseOR](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.bitwiseOR.html#pyspark.sql.Column.bitwiseOR)(other) | Compute bitwise OR of this expression with another expression. |
| [Column.bitwiseXOR](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.bitwiseXOR.html#pyspark.sql.Column.bitwiseXOR)(other) | Compute bitwise XOR of this expression with another expression. |
| [Column.cast](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.cast.html#pyspark.sql.Column.cast)(dataType) | Casts the column into type dataType. |
| [Column.contains](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.contains.html#pyspark.sql.Column.contains)(other) | Contains the other element. Returns Boolean |
| [Column.desc](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.desc.html#pyspark.sql.Column.desc)() | Returns a sort expression based on the descending order of the column. |
| [Column.desc\_nulls\_first](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.desc_nulls_first.html#pyspark.sql.Column.desc_nulls_first)() | Returns a sort expression based on the descending order of the column, and null values appear before non-null values. |
| [Column.desc\_nulls\_last](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.desc_nulls_last.html#pyspark.sql.Column.desc_nulls_last)() | Returns a sort expression based on the descending order of the column, and null values appear after non-null values. |
| [Column.dropFields](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.dropFields.html#pyspark.sql.Column.dropFields)(\*fieldNames) | An expression that drops fields in StructType by name. |
| [Column.endswith](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.endswith.html#pyspark.sql.Column.endswith)(other) | String ends with. |
| [Column.eqNullSafe](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.eqNullSafe.html#pyspark.sql.Column.eqNullSafe)(other) | Equality test that is safe for null values. |
| [Column.getField](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.getField.html#pyspark.sql.Column.getField)(name) | An expression that gets a field by name in a StructType. |
| [Column.getItem](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.getItem.html#pyspark.sql.Column.getItem)(key) | An expression that gets an item at position ordinal out of a list, or gets an item by key out of a dict. |
| [Column.ilike](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.ilike.html#pyspark.sql.Column.ilike)(other) | SQL ILIKE expression (case insensitive LIKE). |
| [Column.isNotNull](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.isNotNull.html#pyspark.sql.Column.isNotNull)() | Returns True if the current expression is NOT null. |
| [Column.isNull](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.isNull.html#pyspark.sql.Column.isNull)() | Returns True if the current expression is null. |
| [Column.isin](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.isin.html#pyspark.sql.Column.isin)(\*cols)  Used to select column values which  are in list of values | A boolean expression that is evaluated to true if the value of this expression is contained by the evaluated values of the arguments. |
| [Column.like](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.like.html#pyspark.sql.Column.like)(other) | SQL like expression. |
| [Column.name](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.name.html#pyspark.sql.Column.name)(\*alias, \*\*kwargs) | name() is an alias for alias(). |
| [Column.otherwise](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.otherwise.html#pyspark.sql.Column.otherwise)(value) | Evaluates a list of conditions and returns one of multiple possible result expressions. |
| [Column.over](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.over.html#pyspark.sql.Column.over)(window) | Define a windowing column. |
| [Column.rlike](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.rlike.html#pyspark.sql.Column.rlike)(other) | SQL RLIKE expression (LIKE with Regex). |
| [Column.startswith](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.startswith.html#pyspark.sql.Column.startswith)(other) | String starts with. |
| [Column.substr](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.substr.html#pyspark.sql.Column.substr)(startPos, length) | Return a [Column](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) which is a substring of the column. |
| [Column.when](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.when.html#pyspark.sql.Column.when)(condition, value) | Evaluates a list of conditions and returns one of multiple possible result expressions. |
| [Column.withField](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.withField.html#pyspark.sql.Column.withField)(fieldName, col) | An expression that adds/replaces a field in StructType by name. |

**Note:**

If any column API function (isNull(), isNotNull() )returns Boolean value then that should be used with filter() or where() to get the data from dataframe instead of select()

* If API column API returns boolean value and used with select then we will get True or False data of pyspark column.
* If used with filter then we will get the dataframe as output after filter operation.

**Selection Column data from Dataframe:**

We can select column data of a dataframe in below ways-

* df.select( column\_object )
* df.select( df.columns\_name )
* df.select (df["col\_name"])
* df.select( col( col1, col2, . . . coln) ) --- to select more than one column data
* df.select ( col("\*")) --- to select all column data from dataframe

**PySpark Column Operators**

PySpark column also provides a way to do arithmetic operations on columns using operators.

|  |  |
| --- | --- |
| ARITHMETIC OPERATION | COMPARISON OPERATION |
| * df.select(df.col1 + df.col2) * df.select(df.col1 - df.col2) * df.select(df.col1 \* df.col2) * df.select(df.col1 / df.col2) * df.select(df.col1 % df.col2) | * df.select(df.col2 > df.col3) * df.select(df.col2 < df.col3) * df.select(df.col2 == df.col3)   Here we are not using with filter then will get boolean values of pyspark col type. |

column.astype(new\_dtype)

column.cast(new\_dtype)

these functions are used to change the column data type to new\_type.

Both function works exactly same.

**Example:**

df.select( df.age.cast("string").alias('ages') )

df.select( df.age.astype("string").alias('ages') )

column.between(X,Y)

*Returns a Boolean expression when a column value in between lower limit(X) and upper limit(Y)*

Basically, this this used to get the dataframe or few columns of dataframe when values of a column are in between lower and upper bound.

**Questions:**

Get the dataframe all column data for which value of ID column is between 100 and 300.

**Solution:**

df.filter(df.id.between(100,300)).show()

column.contains( val )

*Checks if a DataFrame column contains value (val)* contains a value specified in this function.

**Questions:**

Get all column from dataframe where fname column contains 'Cruise' in it.

**Solution:**

df.filter(df.fname.contains("Cruise")).show()

column.startswith('val')

column.endswith('val')

Checks if the value of the DataFrame Column starts and ends with a String respectively.

column.isNull

column.isNotNull()

Checks if the DataFrame column has NULL or non NULL values.

column.like()

column.rlike()

Similar to SQL LIKE expression. This is used to filter data based on regular expression.

#**Question**#

From below email ids filter the valid email id.

data = ['buying books at amazom.com', 'rameses@egypt.com', 'matt@t.co', 'narendra@modi.com']

**Solution**:

Convert into dataframe then use rlike method.

data = ['buying books at amazom.com', 'rameses@egypt.com', 'matt@t.co', 'narendra@modi.com']

# Convert the list to DataFrame

#default column name will be value

df = spark.createDataFrame(data, "string")

#define the pattern for valid email id

pattern='\w+@\w+\.com'**# OR pattern='[a-z0-9\_]+@[a-z0-9\_]+\.com'**

df.filter(col('value').rlike(pattern)).show()

column.substr(n1,n2)

this function is used to get the substring of length n2 starting from n1.

**Example:**

Get the data subrting of lenght 2 starting from index 1 from fnamme column.

**Solution:**

df.select(df.fname.substr(1,2).alias("substr"))

**Note:**

* This **gives only alias column or substring column** data because substr function doesn't return boolean value.
* If any function inside the select function gives Boolean value then we will get pyspark column as output of boolean type.

when() & otherwise()

It is similar to SQL Case When, executes sequence of expressions until it matches the condition and returns a value when match.

These two functions are used in conjustion to create a new column based on condition.

pyspark.sql.functions.when(condition, value)

otherwise(value)

Example1:

from pyspark.sql.functions import when

df.select(df.fname,df.lname,when(df.gender=="M","Male") \

.when(df.gender=="F","Female") \

.when(df.gender==None ,"") \

.otherwise(df.gender).alias("new\_gender") \

)

**Example: ---- Good**

when age column data is greater than 4 then assign 1 , when age <3 assign -1to new column else 0

**Solution:**

from pyspark.sql import functions as F

df.select(df.name, F.when(df.age > 4, 1).when(df.age < 3, -1).otherwise(0))

**#Changing datatype of column name#**

We can change the column data type of any column using below function

**column.astype(dtype)**

**column.cast(dtype)**

Both of these function are used to change the column dattype.

**Qurstion:**

Change the data type of Petal\_Length and Sepal\_Length to floast type.

**Solution:**

iris1\_df1.select(iris1\_df1.Petal\_Length.cast("float"), iris1\_df1.Sepal\_Length.cast("float")).show()

**# Question #**

Get all order from order dataframae for whose order\_status is 'CLOSED','COMPLETED','DELIVERED'

Solution:

df.where(df.order\_status.isin('CLOSED','COMPLETED','DELIVERED']))

**#########################################**

**# Datetime Patterns for Formatting and Parsing #**

**#########################################**

In pyspark we have format specifiers that is used to represent the date, time, datetime format represented in string as we have in Python.

[https://spark.apache.org/docs/latest/sql-ref- datetime-pattern.html](https://spark.apache.org/docs/latest/sql-ref-%20datetime-pattern.html)

| **Symbol** | **Meaning** | **Presentation** | **Examples** |
| --- | --- | --- | --- |
| **G** | Era | Text | AD; Anno Domini |
| **Y** | Year | Year | 2020; 20 |
| **D** | day-of-year | number(3) | 189 |
| **M/L** | month-of-year | Month | 7; 07; Jul; July |
| **d** | day-of-month | number(3) | 28 |
| **Q/q** | quarter-of-year | number/text | 3; 03; Q3; 3rd quarter |
| **E** | day-of-week | Text | Tue; Tuesday |
| **F** | aligned day of week in month | number(1) | 3 |
| **A** | am-pm-of-day | am-pm | PM |
| **H** | clock-hour-of-am-pm (1-12) | number(2) | 12 |
| **K** | hour-of-am-pm (0-11) | number(2) | 0 |
| **K** | clock-hour-of-day (1-24) | number(2) | 0 |
| **H** | hour-of-day (0-23) | number(2) | 0 |
| **M** | minute-of-hour | number(2) | 30 |
| **S** | second-of-minute | number(2) | 55 |
| **S** | fraction-of-second | fraction | 978 |
| **V** | time-zone ID | zone-id | America/Los\_Angeles; Z; -08:30 |
| **Z** | time-zone name | zone-name | Pacific Standard Time; PST |
| **O** | localized zone-offset | offset-O | GMT+8; GMT+08:00; UTC-08:00; |
| **X** | zone-offset ‘Z’ for zero | offset-X | Z; -08; -0830; -08:30; -083015; -08:30:15; |
| **X** | zone-offset | offset-x | +0000; -08; -0830; -08:30; -083015; -08:30:15; |
| **Z** | zone-offset | offset-Z | +0000; -0800; -08:00; |
| **‘** | escape for text | delimiter |  |
| **’‘** | single quote | Literal | ’ |
| **[** | optional section start |  |  |
| **]** | optional section end |  |  |

**Note:**

* **'M' or 'L'**: Month number in a year starting from 1. There is no difference between ‘M’ and ‘L’. Month from 1 to 9 are printed without padding.
* **'MM' or 'LL'**: Month number in a year starting from 1. Zero padding is added for month 1-9
* **'MMM':** Short textual representation in the standard form. E.g -- Jan, Feb, Mar ..

**Question: --- good**

Convert the below dataframe column (date\_str\_2) into date type.

data = [("2023-05-18","01 Jan 2010",), ("2023-12-31", "01 Jan 2010",)]

df = spark.createDataFrame(data, ["date\_str\_1", "date\_str\_2"])

+----------+-----------+

|date\_str\_1| date\_str\_2|

+----------+-----------+

|2023-05-18|01 Jan 2010|

|2023-12-31|01 Jan 2010|

+----------+-----------+

Solution:

df=df.withColumn('date\_str\_2',to\_date(df.date\_str\_2,**format='d MMM yyyy'**))

**################################################################################**

**# pyspark.sql.function API #**

**################################################################################**

PySpark SQL provides several built-in standard functions pyspark.sql.functions to work with DataFrame and SQL queries.

All these PySpark SQL Functions return pyspark.sql.Column type.

<https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/functions.html>

All these functions can be divided into below catageory-

* String Functions
* Date & Time Functions
* Collection Functions
* Math Functions
* Aggregate Functions
* Window Functions
* Sort Functions
* Partition Transformation Functions
* UDF Functions and
* Other Misc Functions

**String function:**

There are huge number of functions few most commonly used are-

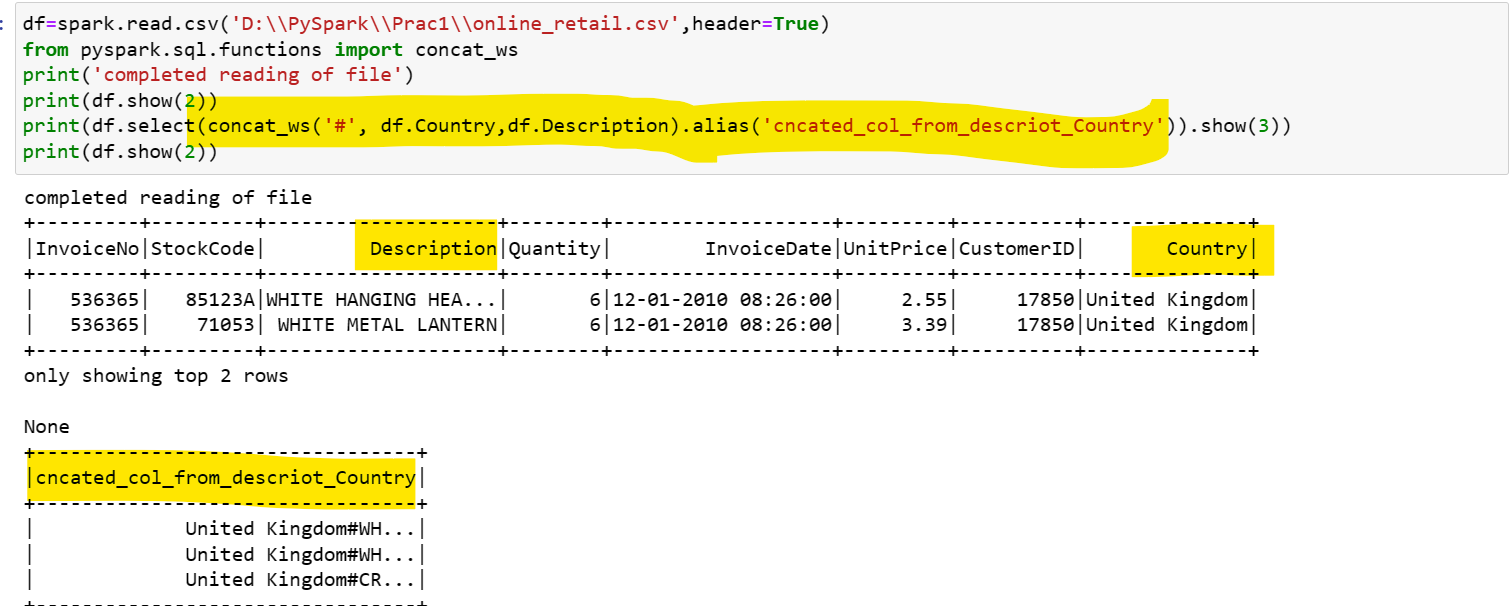
|  |  |
| --- | --- |
| concat\_ws(sep, col1,col2 . . coln) | Concat multiple strings into a single string with a specified separator |
| format\_number(col, d) | Formats the number to ‘#,–#,–#.–’, rounded to d and returns the value in String |
| format\_string(format, \*cols) | Formats the input string to printf-style. |
| length(col) | Returns the length of the input string column. |
| lower(col) | lower() – Converts all characters in a string to lowercase |
| upper(col) | upper() – Convers all characters to uppercase |
| lpad(col, len, pad) | lpad() – Add a specified character as padding on the left side. |
| rpad(col,len,pad) | rpad() – Add a specified character as padding on the right side. |
| ltrim(col) | ltrim() – Removes the space on the left side. |
| rtrim(col) | rtrim() – Removes the spaces on the right side. |
|  |  |
| trim(col) | trim() – Removes the spaces on the both side of given col name. |
| repeat(col, n) | Returns a new string after repeating a column n times |
| split(str, pattern[, limit]) | Splits string by specified patterns |
| substring(str, pos, len) | Returns the substring from te string column. |

**The return type of all above methos is column type.**

concat\_ws()

Concatenates multiple input string columns together into a single string column, using the given separator.

**Example:**



**Example:**

Remove the blanks space from description column.

f=spark.read.csv('D:\\PySpark\\Prac1\\online\_retail.csv',header=True)

print(df.printSchema())

from pyspark.sql.functions import concat\_ws,trim,col,ltrim

print('completed reading of file')

print(df.show(2,truncate =False))

df = df.withColumn('Description', trim('Description'))

OR

df = df.withColumn('Description', trim( col('Description')))

OR

df = df.withColumn('Description', trim(df.Description))

print(df.show(2,truncate =False))

**Datetime Function**

There are many methods that we can see in documentation but commonly used are-

| **Datetime Functions** | **Description** |
| --- | --- |
| [add\_months](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.add_months.html#pyspark.sql.functions.add_months)(given\_date, n) | Returns the date after adding n in month of given date |
| [current\_date](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.current_date.html#pyspark.sql.functions.current_date)() | Returns the current date at the start of query evaluation as a DateType column. |
| [current\_timestamp](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.current_timestamp.html#pyspark.sql.functions.current_timestamp)() | Returns the current timestamp at the start of query evaluation as a TimestampType column. |
| [date\_add](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.date_add.html#pyspark.sql.functions.date_add)(start, days) | Returns the date that is days days after start |
| [date\_format](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.date_format.html#pyspark.sql.functions.date_format)(date, format) | Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument. |
| [date\_sub](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.date_sub.html#pyspark.sql.functions.date_sub)(start, days) | Returns the date that is days days before start |
| [date\_trunc](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.date_trunc.html#pyspark.sql.functions.date_trunc)(format, timestamp) | Returns timestamp truncated to the unit specified by the format. |
| [datediff](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.datediff.html#pyspark.sql.functions.datediff)(end, start) | Returns the number of days from start to end. |
| [dayofmonth](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.dayofmonth.html#pyspark.sql.functions.dayofmonth)(col) | Extract the day of the month of a given date as integer. |
| [dayofweek](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.dayofweek.html#pyspark.sql.functions.dayofweek)(col) | Extract the day of the week of a given date as integer. |
| [dayofyear](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.dayofyear.html#pyspark.sql.functions.dayofyear)(col) | Extract the day of the year of a given date as integer. |
| [second](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.second.html#pyspark.sql.functions.second)(col) | Extract the seconds of a given date as integer. |
| [weekofyear](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.weekofyear.html#pyspark.sql.functions.weekofyear)(col) | Extract the week number of a given date as integer. |
| [year](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.year.html#pyspark.sql.functions.year)(col) | Extract the year of a given date as integer. |
| [quarter](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.quarter.html#pyspark.sql.functions.quarter)(col) | Extract the quarter of a given date as integer. |
| [month](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.month.html#pyspark.sql.functions.month)(col) | Extract the month of a given date as integer. |
| [last\_day](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.last_day.html#pyspark.sql.functions.last_day)(date) | Returns the last day of the month which the given date belongs to. |
| [minute](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.minute.html#pyspark.sql.functions.minute)(col) | Extract the minutes of a given date as integer. |
| [months\_between](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.months_between.html#pyspark.sql.functions.months_between)(date1, date2[, roundOff]) | Returns number of months between dates date1 and date2. |
| [next\_day](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.next_day.html#pyspark.sql.functions.next_day)(date, dayOfWeek) | Returns the first date which is later than the value of the date column. |
| [hour](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.hour.html#pyspark.sql.functions.hour)(col) | Extract the hours of a given date as integer. |
| [make\_date](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.make_date.html#pyspark.sql.functions.make_date)(year, month, day) | Returns a column with a date built from the year, month and day columns. |
| [from\_unixtime](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.from_unixtime.html#pyspark.sql.functions.from_unixtime)(timestamp[, format]) | Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the given format. |
| [unix\_timestamp](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.unix_timestamp.html#pyspark.sql.functions.unix_timestamp)([timestamp, format]) | Convert time string with given pattern (‘yyyy-MM-dd HH:mm:ss’, by default) to Unix time stamp (in seconds), using the default timezone and the default locale, return null if fail. |
| [to\_timestamp](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_timestamp.html#pyspark.sql.functions.to_timestamp)(col[, format]) | Converts a [Column](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) into [pyspark.sql.types.TimestampType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.TimestampType.html" \l "pyspark.sql.types.TimestampType) using the optionally specified format. |
| [to\_date](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_date.html#pyspark.sql.functions.to_date)(col[, format])  return type is col type | Converts a [Column](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) into [pyspark.sql.types.DateType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.DateType.html" \l "pyspark.sql.types.DateType) using the optionally specified format. |
| [trunc](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.trunc.html#pyspark.sql.functions.trunc)(date, format) | Returns date truncated to the unit specified by the format. |
| [from\_utc\_timestamp](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.from_utc_timestamp.html#pyspark.sql.functions.from_utc_timestamp)(timestamp, tz) | This is a common function for databases supporting TIMESTAMP WITHOUT TIMEZONE. |
| [to\_utc\_timestamp](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_utc_timestamp.html#pyspark.sql.functions.to_utc_timestamp)(timestamp, tz) | This is a common function for databases supporting TIMESTAMP WITHOUT TIMEZONE. |
| [window](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.window.html#pyspark.sql.functions.window)(timeColumn, windowDuration[, …]) | Bucketize rows into one or more time windows given a timestamp specifying column. |
| [session\_window](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.session_window.html#pyspark.sql.functions.session_window)(timeColumn, gapDuration) | Generates session window given a timestamp specifying column. |

**Return type of all above method is col type**.

to\_timestamp(col: ColumnOrName, format: col\_format)

Converts a Column into pyspark.sql.types.TimestampType using the optionally specified format.

**################################################################################# GROUPING API #**

**################################################################################**

Grouping API are used to work on group by objects.

We have below methods in PySpark group by objects.

|  |  |
| --- | --- |
| [GroupedData.agg](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.agg.html#pyspark.sql.GroupedData.agg)(\*exprs) | Compute aggregates and returns the result as a [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame). |
| [GroupedData.apply](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.apply.html#pyspark.sql.GroupedData.apply)(udf) | It is an alias of [pyspark.sql.GroupedData.applyInPandas()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.applyInPandas.html#pyspark.sql.GroupedData.applyInPandas); however, it takes a [pyspark.sql.functions.pandas\_udf()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.pandas_udf.html#pyspark.sql.functions.pandas_udf) whereas [pyspark.sql.GroupedData.applyInPandas()](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.applyInPandas.html#pyspark.sql.GroupedData.applyInPandas) takes a Python native function. |
| [GroupedData.applyInPandas](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.applyInPandas.html#pyspark.sql.GroupedData.applyInPandas)(func, schema) | Maps each group of the current [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) using a pandas udf and returns the result as a *DataFrame*. |
| [GroupedData.avg](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.avg.html#pyspark.sql.GroupedData.avg)(\*cols) | Computes average values for each numeric columns for each group. |
| [GroupedData.cogroup](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.cogroup.html#pyspark.sql.GroupedData.cogroup)(other) | Cogroups this group with another group so that we can run cogrouped operations. |
| [GroupedData.count](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.count.html#pyspark.sql.GroupedData.count)() | Counts the number of records for each group. |
| [GroupedData.max](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.max.html#pyspark.sql.GroupedData.max)(\*cols) | Computes the max value for each numeric columns for each group. |
| [GroupedData.mean](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.mean.html#pyspark.sql.GroupedData.mean)(\*cols) | Computes average values for each numeric columns for each group. |
| [GroupedData.min](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.min.html#pyspark.sql.GroupedData.min)(\*cols) | Computes the min value for each numeric column for each group. |
| [GroupedData.pivot](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.pivot.html#pyspark.sql.GroupedData.pivot)(pivot\_col[, values])  rotate/transpose the data from one column  into multiple columns  Check **Example on Pivot** | Pivots a column of the current [DataFrame](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.DataFrame.html#pyspark.sql.DataFrame) and perform the specified aggregation. |
| [GroupedData.sum](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.GroupedData.sum.html#pyspark.sql.GroupedData.sum)(\*cols) | Computes the sum for each numeric columns for each group. |
| [PandasCogroupedOps.applyInPandas](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.PandasCogroupedOps.applyInPandas.html#pyspark.sql.PandasCogroupedOps.applyInPandas)(func, schema) | Applies a function to each cogroup using pandas and returns the result as a *DataFrame*. |

GroupedData.agg(\*exprs, Dict[str, str]])

Compute aggregates and returns the result as a DataFrame.

The available aggregate functions can be:

built-in aggregation functions, such as avg, max, min, sum, count

group aggregate pandas UDFs, created with pyspark.sql.functions.pandas\_udf()

GroupedData.avg(\*cols: str)

Computes average values for each numeric columns for each group.

GroupedData.count()

Counts the number of records for each group.

GroupedData.max(\*cols: str)

Computes the max value for each numeric columns for each group.

**################################################################################**

**# HIVE Integration #**

**################################################################################**

To create hive table, read from hive table or writing dataframe into hive table we need to have hive enables session.

We can create hve enabled session using enableHiveSupport() from SparkSession.

Example:

spark = SparkSession \

.builder \

.appName("SparkByExamples.com") \

.config("spark.sql.warehouse.dir", warehouse\_location) \

.enableHiveSupport() \

.getOrCreate()

######################

# Writing into HIVE table #

#####################

To save a PySpark DataFrame to Hive table use saveAsTable() function or use SQL CREATE statement on top of the temporary view.

**Save DataFrame as Internal Table from PySpark**

dataframe.write.mode('overwrite').saveAsTable('table\_name')

**Save as External Table**

sampleDF.write.mode(SaveMode.Overwrite).option("path", "/path/to/external/table") .saveAsTable("table\_name")

**Using PySpark SQL Temporary View to Save Hive Table**

spark.sql("INSERT INTO TABLE table\_name SELECT \* FROM temp\_table\_view")

**Note:**

If we want to write in any specific database table then we can create database using spark\_session.sql()

spark.sql(" CREATE DATABASE IF NOT EXISTS db\_name ") ----- Creating database.

**Question**:

Create a dataframe and write that dataframe into HIVE table.

**Solution**:

columns = ["id", "name","age","gender"]

# Create DataFrame

data = [(1, "James",30,"M"), (2, "Ann",40,"F"),

(3, "Jeff",41,"M"),(4, "Jennifer",20,"F")]

sampleDF = spark.sparkContext.parallelize(data).toDF(columns)

# Create database

spark.sql("CREATE DATABASE IF NOT EXISTS emp")

# Create Hive Internal table

sampleDF.write.mode('overwrite') \

.saveAsTable("emp.employee")

# Spark read Hive table

df = spark.read.table("emp.employee")

df.show()

**#######################**

**# Reading HIVE table data #**

**#######################**

For reading HIVE table data we can use sql( query ) from pyspark session.

This method reads data from table and return as dataframe.

**Example:**

df = spark.sql("select \* from emp.employee")

df.show()

**################################################################################**

**# User-Defined Functions #**

**################################################################################**

Consider a scenario where a calculated column needs to be created in 10 Hive tables by performing complex operations on one of the existing columns in all of those 10 tables. A user-defined function can be created which performs the operation on its input data using sql and return the result.

Python functions can be registered as Spark User-Defined Functions for two purposes.

1. For being used in Spark SQL statements
2. For being used on Spark DataFrame

**Note:**

We need to first create udf for sql or dataframe then only we can use that udf.

**# UDF for SQL Query #**

For a python function to be used in SQL queries, it needs to be registered using the 'register' function on 'UDFRegistration' class object.

'UDFRegistration' takes sqlContext as an argument to create it’s object.

To create UDF, we pass the following arguments

1. **name** - name of the UDF
2. **func** - python function to be registered as UDF
3. **returnType** - data type of python function return value

Step2:

1. Create the 'UDFRegistration' object ----

udf1 = UDFRegistration(sql\_context )

1. Using 'UDFRegistration' object register the python function as below-

udf1.register(name=udf\_name, python\_func\_name, returnType=return\_type)

Example:

from pyspark.sql.types import FloatType

from pyspark.sql import UDFRegistration

#create the UDFRegirstation object

udf1 = UDFRegistration(sqlContext)

#Now regiter that udf by name new\_fun

udf1.register(name='new\_fun', f=lambda var1: var1+10, returnType=FloatType())

**Example 2: UDF on Temp Tables ---- Using created udf ( new\_fun )**

from pyspark.sql.types import StructType, StructField, FloatType, StringType

iris\_schema = pyspark.sql.types.StructType([

StructField("Sepal\_Length", FloatType(), True),

StructField("Sepal\_Width", FloatType(), True),

StructField("Petal\_Length", FloatType(), True),

StructField("Petal\_Width", FloatType(), True),

StructField("Species", StringType(), True)

])

iris1\_df1 = spark.read.csv(path='./dataset/iris.csv',sep=',',header=True,schema=iris\_schema)

sqlContext.registerDataFrameAsTable(iris1\_df1, "iris\_temp")

sqlContext.sql('select new\_fun( Sepal\_Length ) from iris\_temp').show()

**Example 3: UDF on Hive Tables**

sqlContext.sql('use db')

sqlContext.sql('SELECT new\_fun(Sepal\_Length) FROM iris').show()

**Creating udf using udf decorator ( can be used for dataframe)**

For Data Frame, UDF can be created using 'udf' function in 'pyspark.sql.functions' module.

It takes two arguments.

1. func – python function
2. returnType – a pyspark.sql.types.DataType object

**new\_func\_name = udf ( python\_func, returnType )**

**Note:**

We can create udf using @udf decorator also as belw-

@udf (returnType=return\_type)

def function():

#

return values

**Creating UDF without udf decorator (can be used on dataframe)**

from pyspark.sql.types import FloatType

from pyspark.sql.functions import udf

#UDF is created to add 10 in it’s value by name new\_fun2

new\_fun2 = udf( lambda var1: var1+10, returnType=FloatType() )

**Example 1: ---- Using udf on dataframe.**

from pyspark.sql.types import StructType, StructField, FloatType, StringType

iris\_schema = pyspark.sql.types.StructType([

StructField("Sepal\_Length", FloatType(), True),

StructField("Sepal\_Width", FloatType(), True),

StructField("Petal\_Length", FloatType(), True),

StructField("Petal\_Width", FloatType(), True),

StructField("Species", StringType(), True)

])

iris1\_df1 = spark.read.csv(path='./dataset/iris.csv',sep=',',header=True,schema=iris\_schema)

iris1\_df1.select( new\_fun2 (iris1\_df1.Sepal\_Length ) ).show() # using created udf new\_fun2

**Example 2:**

Create a udf for changing the first letter of each word into upper case for a dataframe.

from pyspark.sql.functions import col, udf

from pyspark.sql.types import StringType

columns = ["Seqno","Name"]

data = [("1", "john jones"),("2", "tracey smith"),("3", "amy sanders")]

df = spark.createDataFrame(data=data,schema=columns)

df.show(truncate=False)

#now create a [ython python to convert into upper case

def convertCase(str):

    resStr=""

    arr = str.split(" ")

    for x in arr:

       resStr= resStr + x[0:1].upper() + x[1:len(x)] + " "

    return resStr

# Converting function to UDF

convertUDF = udf(lambda z: convertCase(z),StringType())

#applyt the UDF on dataframe

df.select(col("Name"),convertUDF(col("Name")).alias("Name\_In\_Upper")).show(truncate=False)

**Note:**

We can create udf with or without decorator and for usages on dataframe but with udf decorator is suggestable.

**Questions: ----- good, always try to solve it**

Create a new column in pyspark dataframe from a list.

|  |  |
| --- | --- |
| **Input dataframe**  +-------+---+--------------+-----+-----+  | name|age| subject|class| fees|  +-------+---+--------------+-----+-----+  | Arun| 16| Maths| 10|12000|  | Aniket| 17|Social Science| 11|15000|  | Ishita| 15| English| 9| 9000|  |Pranjal| 18| Science| 12|18000|  |Vinayak| 18| Computer| 12|18000|  +-------+---+--------------+-----+-----+ | Expected output  fine\_data = [200,300,400,0,500]  +-------+---+--------------+-----+-----+-----------+  | name|age| subject|class| fees|fine\_amt |  +-------+---+--------------+-----+-----+-----------+  | Arun| 16| Maths| 10|12000| 200|  | Aniket| 17|Social Science| 11|15000| 300|  | Ishita| 15| English| 9| 9000| 400|  |Pranjal| 18| Science| 12|18000| 0|  |Vinayak| 18| Computer| 12|18000| 500|  +-------+---+--------------+-----+-----+-----------+ |

Hint:

Create a index column in dataframe

Create a udf which will return a value from list for each value of index column data.

**Solution:**

from pyspark.sql.functions import \*

from pyspark.sql.types import IntegerType, StringType

file\_path=r'D:\Udemy\_PySpark\student\_data.csv'

df=spark.read.csv(file\_path,header=True)

df.show()

#list of data to assign as new column

fine\_data = [200,300,400,0,500]

#create a index column

df=df.withColumn('index',monotonically\_increasing\_id())

df.show()

#create udf, will retunr value from fine\_data list

@udf(returnType=IntegerType())

def my\_udf(x):

    return fine\_data[x-1]

df=df.withColumn('fine\_amount',my\_udf('index'))

df=df.select('name','age','subject','class','fees','fine\_amount')

df.show()

####################

# pyspark.sql.functions #

############# #######

Normal Functions

|  |  |
| --- | --- |
| [**col**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.col.html#pyspark.sql.functions.col)(col) | Returns a [**Column**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) based on the given column name. |
| [**column**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.column.html#pyspark.sql.functions.column)(col) | Returns a [**Column**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) based on the given column name. |
| [**lit**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.lit.html#pyspark.sql.functions.lit)(col) | Creates a [**Column**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) of literal value. |
| [**broadcast**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.broadcast.html#pyspark.sql.functions.broadcast)(df) | Marks a DataFrame as small enough for use in broadcast joins. |
| [**coalesce**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.coalesce.html#pyspark.sql.functions.coalesce)(\*cols) | Returns the first column that is not null. |
| [**input\_file\_name**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.input_file_name.html#pyspark.sql.functions.input_file_name)() | Creates a string column for the file name of the current Spark task. |
| [**isnan**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.isnan.html#pyspark.sql.functions.isnan)(col) | An expression that returns true if the column is NaN. |
| [**isnull**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.isnull.html#pyspark.sql.functions.isnull)(col) | An expression that returns true if the column is null. |
| [**monotonically\_increasing\_id**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.monotonically_increasing_id.html#pyspark.sql.functions.monotonically_increasing_id)()   * creates the integer monotonically   increasing **but not consecutive order**  Example --- assign index to each row  of df | A column that generates monotonically increasing 64-bit integers. |
| [**named\_struct**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.named_struct.html#pyspark.sql.functions.named_struct)(\*cols) | Creates a struct with the given field names and values. |
| [**nanvl**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.nanvl.html#pyspark.sql.functions.nanvl)(col1, col2) | Returns col1 if it is not NaN, or col2 if col1 is NaN. |
| [**rand**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.rand.html#pyspark.sql.functions.rand)([seed]) | Generates a random column with independent and identically distributed (i.i.d.) samples uniformly distributed in [0.0, 1.0). |
| [**randn**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.randn.html#pyspark.sql.functions.randn)([seed]) | Generates a column with independent and identically distributed (i.i.d.) samples from the standard normal distribution. |
| [**spark\_partition\_id**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.spark_partition_id.html#pyspark.sql.functions.spark_partition_id)() | A column for partition ID. |
| [**when**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.when.html#pyspark.sql.functions.when)(condition, value) | Evaluates a list of conditions and returns one of multiple possible result expressions. |
| [**bitwise\_not**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bitwise_not.html#pyspark.sql.functions.bitwise_not)(col) | Computes bitwise not. |
| [**bitwiseNOT**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bitwiseNOT.html#pyspark.sql.functions.bitwiseNOT)(col) | Computes bitwise not. |
| [**expr**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.expr.html#pyspark.sql.functions.expr)(str)  takes SQL expression as a string  argument, executes the expression,  and returns a PySpark Column type.  **Check 'Example expr' this doc** | Parses the expression string into the column that it represents |
| [**greatest**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.greatest.html#pyspark.sql.functions.greatest)(\*cols) | Returns the greatest value of the list of column names, skipping null values. |
| [**least**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.least.html#pyspark.sql.functions.least)(\*cols) | Returns the least value of the list of column names, skipping null values. |

Math Functions

|  |  |
| --- | --- |
| [**sqrt**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sqrt.html#pyspark.sql.functions.sqrt)(col)  calculates the sqrt of col  **return is col type** of computed result | Computes the square root of the specified float value. |
| [**abs**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.abs.html#pyspark.sql.functions.abs)(col) | Computes the absolute value. |
| [**acos**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.acos.html#pyspark.sql.functions.acos)(col) | Computes inverse cosine of the input column. |
| [**acosh**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.acosh.html#pyspark.sql.functions.acosh)(col) | Computes inverse hyperbolic cosine of the input column. |
| [**asin**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.asin.html#pyspark.sql.functions.asin)(col) | Computes inverse sine of the input column. |
| [**asinh**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.asinh.html#pyspark.sql.functions.asinh)(col) | Computes inverse hyperbolic sine of the input column. |
| [**atan**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.atan.html#pyspark.sql.functions.atan)(col) | Compute inverse tangent of the input column. |
| [**atanh**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.atanh.html#pyspark.sql.functions.atanh)(col) | Computes inverse hyperbolic tangent of the input column. |
| [**atan2**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.atan2.html#pyspark.sql.functions.atan2)(col1, col2) | ***New in version 1.4.0.*** |
| [**bin**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bin.html#pyspark.sql.functions.bin)(col) | Returns the string representation of the binary value of the given column. |
| [**cbrt**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.cbrt.html#pyspark.sql.functions.cbrt)(col) | Computes the cube-root of the given value. |
| [**ceil**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.ceil.html#pyspark.sql.functions.ceil)(col) | Computes the ceiling of the given value. |
| [**ceiling**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.ceiling.html#pyspark.sql.functions.ceiling)(col) | Computes the ceiling of the given value. |
| [**conv**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.conv.html#pyspark.sql.functions.conv)(col, fromBase, toBase) | Convert a number in a string column from one base to another. |
| [**cos**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.cos.html#pyspark.sql.functions.cos)(col) | Computes cosine of the input column. |
| [**cosh**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.cosh.html#pyspark.sql.functions.cosh)(col) | Computes hyperbolic cosine of the input column. |
| [**cot**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.cot.html#pyspark.sql.functions.cot)(col) | Computes cotangent of the input column. |
| [**csc**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.csc.html#pyspark.sql.functions.csc)(col) | Computes cosecant of the input column. |
| [**e**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.e.html#pyspark.sql.functions.e)() | Returns Euler’s number. |
| [**exp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.exp.html#pyspark.sql.functions.exp)(col) | Computes the exponential of the given value. |
| [**expm1**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.expm1.html#pyspark.sql.functions.expm1)(col) | Computes the exponential of the given value minus one. |
| [**factorial**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.factorial.html#pyspark.sql.functions.factorial)(col) | Computes the factorial of the given value. |
| [**floor**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.floor.html#pyspark.sql.functions.floor)(col) | Computes the floor of the given value. |
| [**hex**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.hex.html#pyspark.sql.functions.hex)(col) | Computes hex value of the given column, which could be [**pyspark.sql.types.StringType**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.StringType.html#pyspark.sql.types.StringType), [**pyspark.sql.types.BinaryType**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.BinaryType.html#pyspark.sql.types.BinaryType), [**pyspark.sql.types.IntegerType**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.IntegerType.html#pyspark.sql.types.IntegerType) or [**pyspark.sql.types.LongType**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.LongType.html#pyspark.sql.types.LongType). |
| [**unhex**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.unhex.html#pyspark.sql.functions.unhex)(col) | Inverse of hex. |
| [**hypot**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.hypot.html#pyspark.sql.functions.hypot)(col1, col2) | Computes sqrt(a^2 + b^2) without intermediate overflow or underflow. |
| [**ln**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.ln.html#pyspark.sql.functions.ln)(col) | Returns the natural logarithm of the argument. |
| [**log**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.log.html#pyspark.sql.functions.log)(arg1[, arg2]) | Returns the first argument-based logarithm of the second argument. |
| [**log10**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.log10.html#pyspark.sql.functions.log10)(col) | Computes the logarithm of the given value in Base 10. |
| [**log1p**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.log1p.html#pyspark.sql.functions.log1p)(col) | Computes the natural logarithm of the “given value plus one”. |
| [**log2**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.log2.html#pyspark.sql.functions.log2)(col) | Returns the base-2 logarithm of the argument. |
| [**negate**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.negate.html#pyspark.sql.functions.negate)(col) | Returns the negative value. |
| [**negative**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.negative.html#pyspark.sql.functions.negative)(col) | Returns the negative value. |
| [**pi**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.pi.html#pyspark.sql.functions.pi)() | Returns Pi. |
| [**pmod**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.pmod.html#pyspark.sql.functions.pmod)(dividend, divisor) | Returns the positive value of dividend mod divisor. |
| [**positive**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.positive.html#pyspark.sql.functions.positive)(col) | Returns the value. |
| [**pow**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.pow.html#pyspark.sql.functions.pow)(col1, col2)  calculates (col1)\*\*(col2)  col1 and col2 can be number, column etc  **return is col type** of computed result | Returns the value of the first argument raised to the power of the second argument. |
| [**power**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.power.html#pyspark.sql.functions.power)(col1, col2)  --same as aboe | Returns the value of the first argument raised to the power of the second argument. |
| [**rint**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.rint.html#pyspark.sql.functions.rint)(col) | Returns the double value that is closest in value to the argument and is equal to a mathematical integer. |
| [**round**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.round.html#pyspark.sql.functions.round)(col[, scale])  rounds the data of col to given  decimal (scale) places.  **return is col type** of computed result | Round the given value to *scale* decimal places using HALF\_UP rounding mode if *scale* >= 0 or at integral part when *scale* < 0. |
| [**bround**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bround.html#pyspark.sql.functions.bround)(col[, scale]) | Round the given value to *scale* decimal places using HALF\_EVEN rounding mode if *scale* >= 0 or at integral part when *scale* < 0. |
| [**sec**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sec.html#pyspark.sql.functions.sec)(col) | Computes secant of the input column. |
| [**shiftleft**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.shiftleft.html#pyspark.sql.functions.shiftleft)(col, numBits) | Shift the given value numBits left. |
| [**shiftright**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.shiftright.html#pyspark.sql.functions.shiftright)(col, numBits) | (Signed) shift the given value numBits right. |
| [**shiftrightunsigned**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.shiftrightunsigned.html#pyspark.sql.functions.shiftrightunsigned)(col, numBits) | Unsigned shift the given value numBits right. |
| [**sign**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sign.html#pyspark.sql.functions.sign)(col) | Computes the signum of the given value. |
| [**signum**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.signum.html#pyspark.sql.functions.signum)(col) | Computes the signum of the given value. |
| [**sin**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sin.html#pyspark.sql.functions.sin)(col) | Computes sine of the input column. |
| [**sinh**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sinh.html#pyspark.sql.functions.sinh)(col) | Computes hyperbolic sine of the input column. |
| [**tan**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.tan.html#pyspark.sql.functions.tan)(col) | Computes tangent of the input column. |
| [**tanh**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.tanh.html#pyspark.sql.functions.tanh)(col) | Computes hyperbolic tangent of the input column. |
| [**toDegrees**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.toDegrees.html#pyspark.sql.functions.toDegrees)(col) | ***New in version 1.4.0.*** |
| [**try\_add**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.try_add.html#pyspark.sql.functions.try_add)(left, right) | Returns the sum of *left`and `right* and the result is null on overflow. |
| [**try\_avg**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.try_avg.html#pyspark.sql.functions.try_avg)(col) | Returns the mean calculated from values of a group and the result is null on overflow. |
| [**try\_divide**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.try_divide.html#pyspark.sql.functions.try_divide)(left, right) | Returns *dividend*/*divisor*. |
| [**try\_multiply**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.try_multiply.html#pyspark.sql.functions.try_multiply)(left, right) | Returns *left`\*`right* and the result is null on overflow. |
| [**try\_subtract**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.try_subtract.html#pyspark.sql.functions.try_subtract)(left, right) | Returns *left*-*right* and the result is null on overflow. |
| [**try\_sum**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.try_sum.html#pyspark.sql.functions.try_sum)(col) | Returns the sum calculated from values of a group and the result is null on overflow. |
| [**try\_to\_binary**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.try_to_binary.html#pyspark.sql.functions.try_to_binary)(col[, format]) | This is a special version of *to\_binary* that performs the same operation, but returns a NULL value instead of raising an error if the conversion cannot be performed. |
| [**try\_to\_number**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.try_to_number.html#pyspark.sql.functions.try_to_number)(col, format) | Convert string ‘col’ to a number based on the string format *format*. |
| [**degrees**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.degrees.html#pyspark.sql.functions.degrees)(col) | Converts an angle measured in radians to an approximately equivalent angle measured in degrees. |
| [**toRadians**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.toRadians.html#pyspark.sql.functions.toRadians)(col) | ***New in version 1.4.0.*** |
| [**radians**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.radians.html#pyspark.sql.functions.radians)(col) | Converts an angle measured in degrees to an approximately equivalent angle measured in radians. |
| [**width\_bucket**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.width_bucket.html#pyspark.sql.functions.width_bucket)(v, min, max, numBucket) | Returns the bucket number into which the value of this expression would fall after being evaluated. |

Datetime Functions

|  |  |
| --- | --- |
| [**add\_months**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.add_months.html#pyspark.sql.functions.add_months)(start, months) | Returns the date that is *months* months after *start*. |
| [**convert\_timezone**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.convert_timezone.html#pyspark.sql.functions.convert_timezone)(sourceTz, targetTz, sourceTs) | Converts the timestamp without time zone *sourceTs* from the *sourceTz* time zone to *targetTz*. |
| [**curdate**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.curdate.html#pyspark.sql.functions.curdate)() | Returns the current date at the start of query evaluation as a **DateType** column. |
| [**current\_date**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.current_date.html#pyspark.sql.functions.current_date)() | Returns the current date at the start of query evaluation as a **DateType** column. |
| [**current\_timestamp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.current_timestamp.html#pyspark.sql.functions.current_timestamp)() | Returns the current timestamp at the start of query evaluation as a **TimestampType** column. |
| [**current\_timezone**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.current_timezone.html#pyspark.sql.functions.current_timezone)() | Returns the current session local timezone. |
| [**date\_add**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.date_add.html#pyspark.sql.functions.date_add)(start, days) | Returns the date that is *days* days after *start*. |
| [**date\_diff**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.date_diff.html#pyspark.sql.functions.date_diff)(end, start) | Returns the number of days from *start* to *end*. |
| [**date\_format**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.date_format.html#pyspark.sql.functions.date_format)(date, format) | Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument. |
| [**date\_from\_unix\_date**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.date_from_unix_date.html#pyspark.sql.functions.date_from_unix_date)(days) | Create date from the number of *days* since 1970-01-01. |
| [**date\_sub**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.date_sub.html#pyspark.sql.functions.date_sub)(start, days) | Returns the date that is *days* days before *start*. |
| [**date\_trunc**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.date_trunc.html#pyspark.sql.functions.date_trunc)(format, timestamp) | Returns timestamp truncated to the unit specified by the format. |
| [**dateadd**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.dateadd.html#pyspark.sql.functions.dateadd)(start, days) | Returns the date that is *days* days after *start*. |
| [**datediff**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.datediff.html#pyspark.sql.functions.datediff)(end, start) | Returns the number of days from *start* to *end*. |
| [**day**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.day.html#pyspark.sql.functions.day)(col) | Extract the day of the month of a given date/timestamp as integer. |
| [**date\_part**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.date_part.html#pyspark.sql.functions.date_part)(field, source) | Extracts a part of the date/timestamp or interval source. |
| [**datepart**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.datepart.html#pyspark.sql.functions.datepart)(field, source) | Extracts a part of the date/timestamp or interval source. |
| [**dayofmonth**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.dayofmonth.html#pyspark.sql.functions.dayofmonth)(col) | Extract the day of the month of a given date/timestamp as integer. |
| [**dayofweek**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.dayofweek.html#pyspark.sql.functions.dayofweek)(col) | Extract the day of the week of a given date/timestamp as integer. |
| [**dayofyear**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.dayofyear.html#pyspark.sql.functions.dayofyear)(col) | Extract the day of the year of a given date/timestamp as integer. |
| [**extract**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.extract.html#pyspark.sql.functions.extract)(field, source) | Extracts a part of the date/timestamp or interval source. |
| [**second**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.second.html#pyspark.sql.functions.second)(col) | Extract the seconds of a given date as integer. |
| [**weekofyear**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.weekofyear.html#pyspark.sql.functions.weekofyear)(col) | Extract the week number of a given date as integer. |
| [**year**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.year.html#pyspark.sql.functions.year)(col) | Extract the year of a given date/timestamp as integer. |
| [**quarter**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.quarter.html#pyspark.sql.functions.quarter)(col) | Extract the quarter of a given date/timestamp as integer. |
| [**month**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.month.html#pyspark.sql.functions.month)(col) | Extract the month of a given date/timestamp as integer. |
| [**last\_day**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.last_day.html#pyspark.sql.functions.last_day)(date) | Returns the last day of the month which the given date belongs to. |
| [**localtimestamp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.localtimestamp.html#pyspark.sql.functions.localtimestamp)() | Returns the current timestamp without time zone at the start of query evaluation as a timestamp without time zone column. |
| [**make\_dt\_interval**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.make_dt_interval.html#pyspark.sql.functions.make_dt_interval)([days, hours, mins, secs]) | Make DayTimeIntervalType duration from days, hours, mins and secs. |
| [**make\_interval**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.make_interval.html#pyspark.sql.functions.make_interval)([years, months, weeks, days, …]) | Make interval from years, months, weeks, days, hours, mins and secs. |
| [**make\_timestamp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.make_timestamp.html#pyspark.sql.functions.make_timestamp)(years, months, days, hours, …) | Create timestamp from years, months, days, hours, mins, secs and timezone fields. |
| [**make\_timestamp\_ltz**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.make_timestamp_ltz.html#pyspark.sql.functions.make_timestamp_ltz)(years, months, days, …) | Create the current timestamp with local time zone from years, months, days, hours, mins, secs and timezone fields. |
| [**make\_timestamp\_ntz**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.make_timestamp_ntz.html#pyspark.sql.functions.make_timestamp_ntz)(years, months, days, …) | Create local date-time from years, months, days, hours, mins, secs fields. |
| [**make\_ym\_interval**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.make_ym_interval.html#pyspark.sql.functions.make_ym_interval)([years, months]) | Make year-month interval from years, months. |
| [**minute**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.minute.html#pyspark.sql.functions.minute)(col) | Extract the minutes of a given timestamp as integer. |
| [**months\_between**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.months_between.html#pyspark.sql.functions.months_between)(date1, date2[, roundOff]) | Returns number of months between dates date1 and date2. |
| [**next\_day**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.next_day.html#pyspark.sql.functions.next_day)(date, dayOfWeek) | Returns the first date which is later than the value of the date column based on second *week day* argument. |
| [**hour**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.hour.html#pyspark.sql.functions.hour)(col) | Extract the hours of a given timestamp as integer. |
| [**make\_date**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.make_date.html#pyspark.sql.functions.make_date)(year, month, day) | Returns a column with a date built from the year, month and day columns. |
| [**now**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.now.html#pyspark.sql.functions.now)() | Returns the current timestamp at the start of query evaluation. |
| [**from\_unixtime**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.from_unixtime.html#pyspark.sql.functions.from_unixtime)(timestamp[, format]) | Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the given format. |
| [**unix\_timestamp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.unix_timestamp.html#pyspark.sql.functions.unix_timestamp)([timestamp, format]) | Convert time string with given pattern (‘yyyy-MM-dd HH:mm:ss’, by default) to Unix time stamp (in seconds), using the default timezone and the default locale, returns null if failed. |
| [**to\_unix\_timestamp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_unix_timestamp.html#pyspark.sql.functions.to_unix_timestamp)(timestamp[, format]) | Returns the UNIX timestamp of the given time. |
| [**to\_timestamp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_timestamp.html#pyspark.sql.functions.to_timestamp)(col[, format]) | Converts a [**Column**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) into **[pyspark.sql.types.TimestampType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.TimestampType.html" \l "pyspark.sql.types.TimestampType" \o "pyspark.sql.types.TimestampType)** using the optionally specified format. |
| [**to\_timestamp\_ltz**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_timestamp_ltz.html#pyspark.sql.functions.to_timestamp_ltz)(timestamp[, format]) | Parses the *timestamp* with the *format* to a timestamp without time zone. |
| [**to\_timestamp\_ntz**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_timestamp_ntz.html#pyspark.sql.functions.to_timestamp_ntz)(timestamp[, format]) | Parses the *timestamp* with the *format* to a timestamp without time zone. |
| [**to\_date**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_date.html#pyspark.sql.functions.to_date)(col[, format]) | Converts a [**Column**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) into **[pyspark.sql.types.DateType](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.types.DateType.html" \l "pyspark.sql.types.DateType" \o "pyspark.sql.types.DateType)** using the optionally specified format. |
| [**trunc**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.trunc.html#pyspark.sql.functions.trunc)(date, format) | Returns date truncated to the unit specified by the format. |
| [**from\_utc\_timestamp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.from_utc_timestamp.html#pyspark.sql.functions.from_utc_timestamp)(timestamp, tz) | This is a common function for databases supporting TIMESTAMP WITHOUT TIMEZONE. |
| [**to\_utc\_timestamp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_utc_timestamp.html#pyspark.sql.functions.to_utc_timestamp)(timestamp, tz) | This is a common function for databases supporting TIMESTAMP WITHOUT TIMEZONE. |
| [**weekday**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.weekday.html#pyspark.sql.functions.weekday)(col) | Returns the day of the week for date/timestamp (0 = Monday, 1 = Tuesday, …, 6 = Sunday). |
| [**window**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.window.html#pyspark.sql.functions.window)(timeColumn, windowDuration[, …]) | Bucketize rows into one or more time windows given a timestamp specifying column. |
| [**session\_window**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.session_window.html#pyspark.sql.functions.session_window)(timeColumn, gapDuration) | Generates session window given a timestamp specifying column. |
| [**timestamp\_micros**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.timestamp_micros.html#pyspark.sql.functions.timestamp_micros)(col) | Creates timestamp from the number of microseconds since UTC epoch. |
| [**timestamp\_millis**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.timestamp_millis.html#pyspark.sql.functions.timestamp_millis)(col) | Creates timestamp from the number of milliseconds since UTC epoch. |
| [**timestamp\_seconds**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.timestamp_seconds.html#pyspark.sql.functions.timestamp_seconds)(col) | Converts the number of seconds from the Unix epoch (1970-01-01T00:00:00Z) to a timestamp. |
| [**try\_to\_timestamp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.try_to_timestamp.html#pyspark.sql.functions.try_to_timestamp)(col[, format]) | Parses the *col* with the *format* to a timestamp. |
| [**unix\_date**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.unix_date.html#pyspark.sql.functions.unix_date)(col) | Returns the number of days since 1970-01-01. |
| [**unix\_micros**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.unix_micros.html#pyspark.sql.functions.unix_micros)(col) | Returns the number of microseconds since 1970-01-01 00:00:00 UTC. |
| [**unix\_millis**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.unix_millis.html#pyspark.sql.functions.unix_millis)(col) | Returns the number of milliseconds since 1970-01-01 00:00:00 UTC. |
| [**unix\_seconds**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.unix_seconds.html#pyspark.sql.functions.unix_seconds)(col) | Returns the number of seconds since 1970-01-01 00:00:00 UTC. |
| [**window\_time**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.window_time.html#pyspark.sql.functions.window_time)(windowColumn) | Computes the event time from a window column. |

Collection Functions

|  |  |
| --- | --- |
| [**array**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array.html#pyspark.sql.functions.array)(\*cols)  Creates array by taking the values of 'cols'  of each row and return as pyspark column  **Check Example 40** | Creates a new array column. |
| [**array\_contains**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_contains.html#pyspark.sql.functions.array_contains)(col, value) | Collection function: returns null if the array is null, true if the array contains the given value, and false otherwise. |
| [**arrays\_overlap**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.arrays_overlap.html#pyspark.sql.functions.arrays_overlap)(a1, a2) | Collection function: returns true if the arrays contain any common non-null element; if not, returns null if both the arrays are non-empty and any of them contains a null element; returns false otherwise. |
| [**array\_join**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_join.html#pyspark.sql.functions.array_join)(col, delimiter[, null\_replacement]) | Concatenates the elements of *column* using the *delimiter*. |
| [**create\_map**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.create_map.html#pyspark.sql.functions.create_map)(\*cols)  Used to create MapType() column from given columns (cols).  Cols as – key1,value1, key2, value2..  **For each input argument of create\_map()-**  **First argument : --- key**  **Second argument : -- value of first key (First argument)**  **Second argument : --- key**  **Second argument : -- value of first key (Second argument)**  And so on | Creates a new map column.  **Check Example/ Seach for**  **Create MapType from dataframe column** |
| [**slice**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.slice.html#pyspark.sql.functions.slice)(x, start, length) | Collection function: returns an array containing all the elements in *x* from index *start* (array indices start at 1, or from the end if *start* is negative) with the specified *length*. |
| [**concat**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.concat.html#pyspark.sql.functions.concat)(\*cols) | Concatenates multiple input columns together into a single column. |
| [**array\_position**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_position.html#pyspark.sql.functions.array_position)(col, value) | Collection function: Locates the position of the first occurrence of the given value in the given array. |
| [**element\_at**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.element_at.html#pyspark.sql.functions.element_at)(col, extraction) | Collection function: Returns element of array at given index in *extraction* if col is array. |
| [**array\_append**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_append.html#pyspark.sql.functions.array_append)(col, value) | Collection function: returns an array of the elements in col1 along with the added element in col2 at the last of the array. |
| [**array\_size**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_size.html#pyspark.sql.functions.array_size)(col) | Returns the total number of elements in the array. |
| [**array\_sort**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_sort.html#pyspark.sql.functions.array_sort)(col[, comparator]) | Collection function: sorts the input array in ascending order. |
| [**array\_insert**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_insert.html#pyspark.sql.functions.array_insert)(arr, pos, value) | Collection function: adds an item into a given array at a specified array index. |
| [**array\_remove**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_remove.html#pyspark.sql.functions.array_remove)(col, element) | Collection function: Remove all elements that equal to element from the given array. |
| [**array\_prepend**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_prepend.html#pyspark.sql.functions.array_prepend)(col, value) | Collection function: Returns an array containing element as well as all elements from array. |
| [**array\_distinct**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_distinct.html#pyspark.sql.functions.array_distinct)(col) | Collection function: removes duplicate values from the array. |
| [**array\_intersect**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_intersect.html#pyspark.sql.functions.array_intersect)(col1, col2) | Collection function: returns an array of the elements in the intersection of col1 and col2, without duplicates. |
| [**array\_union**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_union.html#pyspark.sql.functions.array_union)(col1, col2) | Collection function: returns an array of the elements in the union of col1 and col2, without duplicates. |
| [**array\_except**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_except.html#pyspark.sql.functions.array_except)(col1, col2) | Collection function: returns an array of the elements in col1 but not in col2, without duplicates. |
| [**array\_compact**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_compact.html#pyspark.sql.functions.array_compact)(col) | Collection function: removes null values from the array. |
| [**transform**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.transform.html#pyspark.sql.functions.transform)(col, f) | Returns an array of elements after applying a transformation to each element in the input array. |
| [**exists**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.exists.html#pyspark.sql.functions.exists)(col, f) | Returns whether a predicate holds for one or more elements in the array. |
| [**forall**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.forall.html#pyspark.sql.functions.forall)(col, f) | Returns whether a predicate holds for every element in the array. |
| [**filter**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.filter.html#pyspark.sql.functions.filter)(col, f) | Returns an array of elements for which a predicate holds in a given array. |
| [**aggregate**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.aggregate.html#pyspark.sql.functions.aggregate)(col, initialValue, merge[, finish]) | Applies a binary operator to an initial state and all elements in the array, and reduces this to a single state. |
| [**zip\_with**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.zip_with.html#pyspark.sql.functions.zip_with)(left, right, f) | Merge two given arrays, element-wise, into a single array using a function. |
| [**transform\_keys**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.transform_keys.html#pyspark.sql.functions.transform_keys)(col, f) | Applies a function to every key-value pair in a map and returns a map with the results of those applications as the new keys for the pairs. |
| [**transform\_values**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.transform_values.html#pyspark.sql.functions.transform_values)(col, f) | Applies a function to every key-value pair in a map and returns a map with the results of those applications as the new values for the pairs. |
| [**map\_filter**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.map_filter.html#pyspark.sql.functions.map_filter)(col, f) | Returns a map whose key-value pairs satisfy a predicate. |
| [**map\_from\_arrays**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.map_from_arrays.html#pyspark.sql.functions.map_from_arrays)(col1, col2) | Creates a new map from two arrays. |
| [**map\_zip\_with**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.map_zip_with.html#pyspark.sql.functions.map_zip_with)(col1, col2, f) | Merge two given maps, key-wise into a single map using a function. |
| [**explode**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.explode.html#pyspark.sql.functions.explode)(col)  **if for MapType() column:**  new col name will be 'key' & 'value' --default  **alias('my\_key','my\_val')**  --- to change the col name  **if for ArrayType() column:**  ---- Creates only one column  **explode(col1) don’t create row for value**  **of row for each val from dict/list.**  **Example:**  Search **'explode on maptype col'** | Returns a new row for each element in the given array or map.  **search “explode() vs explode\_outer()”** |
| [**explode\_outer**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.explode_outer.html#pyspark.sql.functions.explode_outer)(col)  **Same as explode() but it create row**  **even for null value for each value map/list.**  **search “explode() vs explode\_outer()”** | Returns a new row for each element in the given array or map. |
| [**posexplode**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.posexplode.html#pyspark.sql.functions.posexplode)(col) | Returns a new row for each element with position in the given array or map. |
| [**posexplode\_outer**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.posexplode_outer.html#pyspark.sql.functions.posexplode_outer)(col) | Returns a new row for each element with position in the given array or map. |
| [**inline**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.inline.html#pyspark.sql.functions.inline)(col) | Explodes an array of structs into a table. |
| [**inline\_outer**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.inline_outer.html#pyspark.sql.functions.inline_outer)(col) | Explodes an array of structs into a table. |
| [**get**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.get.html#pyspark.sql.functions.get)(col, index) | Collection function: Returns element of array at given (0-based) index. |
| [**get\_json\_object**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.get_json_object.html#pyspark.sql.functions.get_json_object)(col, path) | Extracts json object from a json string based on json *path* specified, and returns json string of the extracted json object. |
| [**json\_tuple**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.json_tuple.html#pyspark.sql.functions.json_tuple)(col, \*fields)  extract the elements from JSON  column from df  and create the new column in df for  each value of fileds (list type)  each col/key name from json data type  column 'col' given as fields will be  created a new col in result dataframe.  **Search -- 'Example on json\_tuple'** | Creates a new row for a json column according to the given field names. |
| [**from\_json**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.from_json.html#pyspark.sql.functions.from_json)(col, schema[, options])  Converts JSON string (which is string type)  into Struct type or Map type.  **Check Example –**  #Example # Assigning/creation a  dict\_value to pyspark column | Parses a column containing a JSON string into a **MapType** with **StringType** as keys type, **StructType** or **ArrayType** with the specified schema. |
| [**schema\_of\_json**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.schema_of_json.html#pyspark.sql.functions.schema_of_json)(json[, options]) | Parses a JSON string and infers its schema in DDL format. |
| [**to\_json**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_json.html#pyspark.sql.functions.to_json)(col[, options])  convert DataFrame columns (  MapType or Struct type)  to JSON string (python string)  Opposite of above  #Example # to\_json | Converts a column containing a **StructType**, **ArrayType** or a **MapType** into a JSON string. |
| [**json\_array\_length**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.json_array_length.html#pyspark.sql.functions.json_array_length)(col) | Returns the number of elements in the outermost JSON array. |
| [**json\_object\_keys**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.json_object_keys.html#pyspark.sql.functions.json_object_keys)(col) | Returns all the keys of the outermost JSON object as an array. |
| [**size**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.size.html#pyspark.sql.functions.size)(col) | Collection function: returns the length of the array or map stored in the column. |
| [**cardinality**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.cardinality.html#pyspark.sql.functions.cardinality)(col) | Collection function: returns the length of the array or map stored in the column. |
| [**struct**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.struct.html#pyspark.sql.functions.struct)(\*cols) | Creates a new struct column. |
| [**sort\_array**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sort_array.html#pyspark.sql.functions.sort_array)(col[, asc]) | Collection function: sorts the input array in ascending or descending order according to the natural ordering of the array elements. |
| [**array\_max**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_max.html#pyspark.sql.functions.array_max)(col) | Collection function: returns the maximum value of the array. |
| [**array\_min**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_min.html#pyspark.sql.functions.array_min)(col) | Collection function: returns the minimum value of the array. |
| [**shuffle**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.shuffle.html#pyspark.sql.functions.shuffle)(col) | Collection function: Generates a random permutation of the given array. |
| [**reverse**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.reverse.html#pyspark.sql.functions.reverse)(col) | Collection function: returns a reversed string or an array with reverse order of elements. |
| [**flatten**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.flatten.html#pyspark.sql.functions.flatten)(col) | Collection function: creates a single array from an array of arrays. |
| [**sequence**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sequence.html#pyspark.sql.functions.sequence)(start, stop[, step]) | Generate a sequence of integers from *start* to *stop*, incrementing by *step*. |
| [**array\_repeat**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_repeat.html#pyspark.sql.functions.array_repeat)(col, count) | Collection function: creates an array containing a column repeated count times. |
| [**map\_contains\_key**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.map_contains_key.html#pyspark.sql.functions.map_contains_key)(col, value) | Returns true if the map contains the key. |
| [**map\_keys**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.map_keys.html#pyspark.sql.functions.map_keys)(col) | Collection function: Returns an unordered array containing the keys of the map. |
| [**map\_values**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.map_values.html#pyspark.sql.functions.map_values)(col) | Collection function: Returns an unordered array containing the values of the map. |
| [**map\_entries**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.map_entries.html#pyspark.sql.functions.map_entries)(col) | Collection function: Returns an unordered array of all entries in the given map. |
| [**map\_from\_entries**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.map_from_entries.html#pyspark.sql.functions.map_from_entries)(col) | Collection function: Converts an array of entries (key value struct types) to a map of values. |
| [**arrays\_zip**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.arrays_zip.html#pyspark.sql.functions.arrays_zip)(\*cols) | Collection function: Returns a merged array of structs in which the N-th struct contains all N-th values of input arrays. |
| [**map\_concat**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.map_concat.html#pyspark.sql.functions.map_concat)(\*cols) | Returns the union of all the given maps. |
| [**from\_csv**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.from_csv.html#pyspark.sql.functions.from_csv)(col, schema[, options]) | Parses a column containing a CSV string to a row with the specified schema. |
| [**schema\_of\_csv**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.schema_of_csv.html#pyspark.sql.functions.schema_of_csv)(csv[, options]) | Parses a CSV string and infers its schema in DDL format. |
| [**str\_to\_map**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.str_to_map.html#pyspark.sql.functions.str_to_map)(text[, pairDelim, keyValueDelim]) | Creates a map after splitting the text into key/value pairs using delimiters. |
| [**to\_csv**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_csv.html#pyspark.sql.functions.to_csv)(col[, options]) | Converts a column containing a **StructType** into a CSV string. |
| [**try\_element\_at**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.try_element_at.html#pyspark.sql.functions.try_element_at)(col, extraction) | (array, index) - Returns element of array at given (1-based) index. |

Partition Transformation Functions

|  |  |
| --- | --- |
| [**years**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.years.html#pyspark.sql.functions.years)(col) | Partition transform function: A transform for timestamps and dates to partition data into years. |
| [**months**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.months.html#pyspark.sql.functions.months)(col) | Partition transform function: A transform for timestamps and dates to partition data into months. |
| [**days**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.days.html#pyspark.sql.functions.days)(col) | Partition transform function: A transform for timestamps and dates to partition data into days. |
| [**hours**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.hours.html#pyspark.sql.functions.hours)(col) | Partition transform function: A transform for timestamps to partition data into hours. |
| [**bucket**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bucket.html#pyspark.sql.functions.bucket)(numBuckets, col) | Partition transform function: A transform for any type that partitions by a hash of the input column. |

Aggregate Functions

|  |  |
| --- | --- |
| [**any\_value**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.any_value.html#pyspark.sql.functions.any_value)(col[, ignoreNulls]) | Returns some value of *col* for a group of rows. |
| [**approxCountDistinct**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.approxCountDistinct.html#pyspark.sql.functions.approxCountDistinct)(col[, rsd]) | ***New in version 1.3.0.*** |
| [**approx\_count\_distinct**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.approx_count_distinct.html#pyspark.sql.functions.approx_count_distinct)(col[, rsd]) | Aggregate function: returns a new [**Column**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) for approximate distinct count of column *col*. |
| [**approx\_percentile**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.approx_percentile.html#pyspark.sql.functions.approx_percentile)(col, percentage[, accuracy]) | Returns the approximate *percentile* of the numeric column *col* which is the smallest value in the ordered *col* values (sorted from least to greatest) such that no more than *percentage* of *col* values is less than the value or equal to that value. |
| [**array\_agg**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.array_agg.html#pyspark.sql.functions.array_agg)(col) | Aggregate function: returns a list of objects with duplicates. |
| [**avg**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.avg.html#pyspark.sql.functions.avg)(col) | Aggregate function: returns the average of the values in a group. |
| [**bit\_and**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bit_and.html#pyspark.sql.functions.bit_and)(col) | Aggregate function: returns the bitwise AND of all non-null input values, or null if none. |
| [**bit\_or**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bit_or.html#pyspark.sql.functions.bit_or)(col) | Aggregate function: returns the bitwise OR of all non-null input values, or null if none. |
| [**bit\_xor**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bit_xor.html#pyspark.sql.functions.bit_xor)(col) | Aggregate function: returns the bitwise XOR of all non-null input values, or null if none. |
| [**bool\_and**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bool_and.html#pyspark.sql.functions.bool_and)(col) | Aggregate function: returns true if all values of *col* are true. |
| [**bool\_or**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bool_or.html#pyspark.sql.functions.bool_or)(col) | Aggregate function: returns true if at least one value of *col* is true. |
| [**collect\_list**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.collect_list.html#pyspark.sql.functions.collect_list)(col) | Aggregate function: returns a list of objects with duplicates. |
| [**collect\_set**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.collect_set.html#pyspark.sql.functions.collect_set)(col) | Aggregate function: returns a set of objects with duplicate elements eliminated. |
| [**corr**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.corr.html#pyspark.sql.functions.corr)(col1, col2) | Returns a new [**Column**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) for the Pearson Correlation Coefficient for col1 and col2. |
| [**count**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.count.html#pyspark.sql.functions.count)(col) | Aggregate function: returns the number of items in a group. |
| [**count\_distinct**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.count_distinct.html#pyspark.sql.functions.count_distinct)(col, \*cols) | Returns a new **Column** for distinct count of col or cols. |
| [**countDistinct**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.countDistinct.html#pyspark.sql.functions.countDistinct)(col, \*cols) | Returns a new [**Column**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) for distinct count of col or cols. |
| [**count\_min\_sketch**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.count_min_sketch.html#pyspark.sql.functions.count_min_sketch)(col, eps, confidence, seed) | Returns a count-min sketch of a column with the given esp, confidence and seed. |
| [**count\_if**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.count_if.html#pyspark.sql.functions.count_if)(col) | Returns the number of *TRUE* values for the *col*. |
| [**covar\_pop**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.covar_pop.html#pyspark.sql.functions.covar_pop)(col1, col2) | Returns a new [**Column**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) for the population covariance of col1 and col2. |
| [**covar\_samp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.covar_samp.html#pyspark.sql.functions.covar_samp)(col1, col2) | Returns a new [**Column**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Column.html#pyspark.sql.Column) for the sample covariance of col1 and col2. |
| [**every**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.every.html#pyspark.sql.functions.every)(col) | Aggregate function: returns true if all values of *col* are true. |
| [**first**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.first.html#pyspark.sql.functions.first)(col[, ignorenulls]) | Aggregate function: returns the first value in a group. |
| [**first\_value**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.first_value.html#pyspark.sql.functions.first_value)(col[, ignoreNulls]) | Returns the first value of *col* for a group of rows. |
| [**grouping**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.grouping.html#pyspark.sql.functions.grouping)(col) | Aggregate function: indicates whether a specified column in a GROUP BY list is aggregated or not, returns 1 for aggregated or 0 for not aggregated in the result set. |
| [**grouping\_id**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.grouping_id.html#pyspark.sql.functions.grouping_id)(\*cols) | Aggregate function: returns the level of grouping, equals to |
| [**histogram\_numeric**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.histogram_numeric.html#pyspark.sql.functions.histogram_numeric)(col, nBins) | Computes a histogram on numeric ‘col’ using nb bins. |
| [**hll\_sketch\_agg**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.hll_sketch_agg.html#pyspark.sql.functions.hll_sketch_agg)(col[, lgConfigK]) | Aggregate function: returns the updatable binary representation of the Datasketches HllSketch configured with lgConfigK arg. |
| [**hll\_union\_agg**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.hll_union_agg.html#pyspark.sql.functions.hll_union_agg)(col[, allowDifferentLgConfigK]) | Aggregate function: returns the updatable binary representation of the Datasketches HllSketch, generated by merging previously created Datasketches HllSketch instances via a Datasketches Union instance. |
| [**kurtosis**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.kurtosis.html#pyspark.sql.functions.kurtosis)(col) | Aggregate function: returns the kurtosis of the values in a group. |
| [**last**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.last.html#pyspark.sql.functions.last)(col[, ignorenulls]) | Aggregate function: returns the last value in a group. |
| [**last\_value**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.last_value.html#pyspark.sql.functions.last_value)(col[, ignoreNulls]) | Returns the last value of *col* for a group of rows. |
| [**max**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.max.html#pyspark.sql.functions.max)(col) | Aggregate function: returns the maximum value of the expression in a group. |
| [**max\_by**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.max_by.html#pyspark.sql.functions.max_by)(col, ord) | Returns the value associated with the maximum value of ord. |
| [**mean**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.mean.html#pyspark.sql.functions.mean)(col) | Aggregate function: returns the average of the values in a group. |
| [**median**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.median.html#pyspark.sql.functions.median)(col) | Returns the median of the values in a group. |
| [**min**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.min.html#pyspark.sql.functions.min)(col) | Aggregate function: returns the minimum value of the expression in a group. |
| [**min\_by**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.min_by.html#pyspark.sql.functions.min_by)(col, ord) | Returns the value associated with the minimum value of ord. |
| [**mode**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.mode.html#pyspark.sql.functions.mode)(col) | Returns the most frequent value in a group. |
| [**percentile**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.percentile.html#pyspark.sql.functions.percentile)(col, percentage[, frequency]) | Returns the exact percentile(s) of numeric column *expr* at the given percentage(s) with value range in [0.0, 1.0]. |
| [**percentile\_approx**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.percentile_approx.html#pyspark.sql.functions.percentile_approx)(col, percentage[, accuracy]) | Returns the approximate *percentile* of the numeric column *col* which is the smallest value in the ordered *col* values (sorted from least to greatest) such that no more than *percentage* of *col* values is less than the value or equal to that value. |
| [**product**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.product.html#pyspark.sql.functions.product)(col) | Aggregate function: returns the product of the values in a group. |
| [**reduce**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.reduce.html#pyspark.sql.functions.reduce)(col, initialValue, merge[, finish]) | Applies a binary operator to an initial state and all elements in the array, and reduces this to a single state. |
| [**regr\_avgx**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regr_avgx.html#pyspark.sql.functions.regr_avgx)(y, x) | Aggregate function: returns the average of the independent variable for non-null pairs in a group, where *y* is the dependent variable and *x* is the independent variable. |
| [**regr\_avgy**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regr_avgy.html#pyspark.sql.functions.regr_avgy)(y, x) | Aggregate function: returns the average of the dependent variable for non-null pairs in a group, where *y* is the dependent variable and *x* is the independent variable. |
| [**regr\_count**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regr_count.html#pyspark.sql.functions.regr_count)(y, x) | Aggregate function: returns the number of non-null number pairs in a group, where *y* is the dependent variable and *x* is the independent variable. |
| [**regr\_intercept**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regr_intercept.html#pyspark.sql.functions.regr_intercept)(y, x) | Aggregate function: returns the intercept of the univariate linear regression line for non-null pairs in a group, where *y* is the dependent variable and *x* is the independent variable. |
| [**regr\_r2**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regr_r2.html#pyspark.sql.functions.regr_r2)(y, x) | Aggregate function: returns the coefficient of determination for non-null pairs in a group, where *y* is the dependent variable and *x* is the independent variable. |
| [**regr\_slope**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regr_slope.html#pyspark.sql.functions.regr_slope)(y, x) | Aggregate function: returns the slope of the linear regression line for non-null pairs in a group, where *y* is the dependent variable and *x* is the independent variable. |
| [**regr\_sxx**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regr_sxx.html#pyspark.sql.functions.regr_sxx)(y, x) | Aggregate function: returns REGR\_COUNT(y, x) \* VAR\_POP(x) for non-null pairs in a group, where *y* is the dependent variable and *x* is the independent variable. |
| [**regr\_sxy**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regr_sxy.html#pyspark.sql.functions.regr_sxy)(y, x) | Aggregate function: returns REGR\_COUNT(y, x) \* COVAR\_POP(y, x) for non-null pairs in a group, where *y* is the dependent variable and *x* is the independent variable. |
| [**regr\_syy**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regr_syy.html#pyspark.sql.functions.regr_syy)(y, x) | Aggregate function: returns REGR\_COUNT(y, x) \* VAR\_POP(y) for non-null pairs in a group, where *y* is the dependent variable and *x* is the independent variable. |
| [**skewness**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.skewness.html#pyspark.sql.functions.skewness)(col) | Aggregate function: returns the skewness of the values in a group. |
| [**some**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.some.html#pyspark.sql.functions.some)(col) | Aggregate function: returns true if at least one value of *col* is true. |
| [**std**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.std.html#pyspark.sql.functions.std)(col) | Aggregate function: alias for stddev\_samp. |
| [**stddev**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.stddev.html#pyspark.sql.functions.stddev)(col) | Aggregate function: alias for stddev\_samp. |
| [**stddev\_pop**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.stddev_pop.html#pyspark.sql.functions.stddev_pop)(col) | Aggregate function: returns population standard deviation of the expression in a group. |
| [**stddev\_samp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.stddev_samp.html#pyspark.sql.functions.stddev_samp)(col) | Aggregate function: returns the unbiased sample standard deviation of the expression in a group. |
| [**sum**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sum.html#pyspark.sql.functions.sum)(col) | Aggregate function: returns the sum of all values in the expression. |
| [**sum\_distinct**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sum_distinct.html#pyspark.sql.functions.sum_distinct)(col) | Aggregate function: returns the sum of distinct values in the expression. |
| [**sumDistinct**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sumDistinct.html#pyspark.sql.functions.sumDistinct)(col) | Aggregate function: returns the sum of distinct values in the expression. |
| [**var\_pop**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.var_pop.html#pyspark.sql.functions.var_pop)(col) | Aggregate function: returns the population variance of the values in a group. |
| [**var\_samp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.var_samp.html#pyspark.sql.functions.var_samp)(col) | Aggregate function: returns the unbiased sample variance of the values in a group. |
| [**variance**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.variance.html#pyspark.sql.functions.variance)(col) | Aggregate function: alias for var\_samp |
|  |  |

**Note:**

**NULL values are ignored from processing by all the aggregate functions**

Window Functions

|  |  |
| --- | --- |
| [**cume\_dist**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.cume_dist.html#pyspark.sql.functions.cume_dist)() | Window function: returns the cumulative distribution of values within a window partition, i.e. |
| [**dense\_rank**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.dense_rank.html#pyspark.sql.functions.dense_rank)() | Window function: returns the rank of rows within a window partition, without any gaps. |
| [**lag**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.lag.html#pyspark.sql.functions.lag)(col[, offset, default]) | Window function: returns the value that is *offset* rows before the current row, and *default* if there is less than *offset* rows before the current row. |
| [**lead**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.lead.html#pyspark.sql.functions.lead)(col[, offset, default]) | Window function: returns the value that is *offset* rows after the current row, and *default* if there is less than *offset* rows after the current row. |
| [**nth\_value**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.nth_value.html#pyspark.sql.functions.nth_value)(col, offset[, ignoreNulls]) | Window function: returns the value that is the *offset*th row of the window frame (counting from 1), and *null* if the size of window frame is less than *offset* rows. |
| [**ntile**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.ntile.html#pyspark.sql.functions.ntile)(n) | Window function: returns the ntile group id (from 1 to *n* inclusive) in an ordered window partition. |
| [**percent\_rank**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.percent_rank.html#pyspark.sql.functions.percent_rank)() | Window function: returns the relative rank (i.e. |
| [**rank**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.rank.html#pyspark.sql.functions.rank)() | Window function: returns the rank of rows within a window partition. |
| [**row\_number**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.row_number.html#pyspark.sql.functions.row_number)() | Window function: returns a sequential number starting at 1 within a window partition. |

Sort Functions

|  |  |
| --- | --- |
| [**asc**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.asc.html#pyspark.sql.functions.asc)(col) | Returns a sort expression based on the ascending order of the given column name. |
| [**asc\_nulls\_first**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.asc_nulls_first.html#pyspark.sql.functions.asc_nulls_first)(col) | Returns a sort expression based on the ascending order of the given column name, and null values return before non-null values. |
| [**asc\_nulls\_last**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.asc_nulls_last.html#pyspark.sql.functions.asc_nulls_last)(col) | Returns a sort expression based on the ascending order of the given column name, and null values appear after non-null values. |
| [**desc**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.desc.html#pyspark.sql.functions.desc)(col) | Returns a sort expression based on the descending order of the given column name. |
| [**desc\_nulls\_first**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.desc_nulls_first.html#pyspark.sql.functions.desc_nulls_first)(col) | Returns a sort expression based on the descending order of the given column name, and null values appear before non-null values. |
| [**desc\_nulls\_last**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.desc_nulls_last.html#pyspark.sql.functions.desc_nulls_last)(col) | Returns a sort expression based on the descending order of the given column name, and null values appear after non-null values. |

String Functions

|  |  |
| --- | --- |
| [**ascii**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.ascii.html#pyspark.sql.functions.ascii)(col) | Computes the numeric value of the first character of the string column. |
| [**base64**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.base64.html#pyspark.sql.functions.base64)(col) | Computes the BASE64 encoding of a binary column and returns it as a string column. |
| [**bit\_length**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bit_length.html#pyspark.sql.functions.bit_length)(col) | Calculates the bit length for the specified string column. |
| [**btrim**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.btrim.html#pyspark.sql.functions.btrim)(str[, trim]) | Remove the leading and trailing *trim* characters from *str*. |
| [**char**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.char.html#pyspark.sql.functions.char)(col) | Returns the ASCII character having the binary equivalent to *col*. |
| [**character\_length**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.character_length.html#pyspark.sql.functions.character_length)(str) | Returns the character length of string data or number of bytes of binary data. |
| [**char\_length**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.char_length.html#pyspark.sql.functions.char_length)(str) | Returns the character length of string data or number of bytes of binary data. |
| [**concat\_ws**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.concat_ws.html#pyspark.sql.functions.concat_ws)(sep, \*cols) | Concatenates multiple input string columns together into a single string column, using the given separator. |
| [**contains**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.contains.html#pyspark.sql.functions.contains)(left, right) | Returns a boolean. |
| [**decode**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.decode.html#pyspark.sql.functions.decode)(col, charset) | Computes the first argument into a string from a binary using the provided character set (one of ‘US-ASCII’, ‘ISO-8859-1’, ‘UTF-8’, ‘UTF-16BE’, ‘UTF-16LE’, ‘UTF-16’). |
| [**elt**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.elt.html#pyspark.sql.functions.elt)(\*inputs) | Returns the *n*-th input, e.g., returns *input2* when *n* is 2. |
| [**encode**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.encode.html#pyspark.sql.functions.encode)(col, charset) | Computes the first argument into a binary from a string using the provided character set (one of ‘US-ASCII’, ‘ISO-8859-1’, ‘UTF-8’, ‘UTF-16BE’, ‘UTF-16LE’, ‘UTF-16’). |
| [**endswith**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.endswith.html#pyspark.sql.functions.endswith)(str, suffix) | Returns a boolean. |
| [**find\_in\_set**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.find_in_set.html#pyspark.sql.functions.find_in_set)(str, str\_array) | Returns the index (1-based) of the given string (*str*) in the comma-delimited list (*strArray*). |
| [**format\_number**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.format_number.html#pyspark.sql.functions.format_number)(col, d) | Formats the number X to a format like ‘#,–#,–#.–’, rounded to d decimal places with HALF\_EVEN round mode, and returns the result as a string. |
| [**format\_string**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.format_string.html#pyspark.sql.functions.format_string)(format, \*cols) | Formats the arguments in printf-style and returns the result as a string column. |
| [**ilike**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.ilike.html#pyspark.sql.functions.ilike)(str, pattern[, escapeChar]) | Returns true if str matches *pattern* with *escape* case-insensitively, null if any arguments are null, false otherwise. |
| [**initcap**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.initcap.html#pyspark.sql.functions.initcap)(col) | Translate the first letter of each word to upper case in the sentence. |
| [**instr**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.instr.html#pyspark.sql.functions.instr)(str, substr) | Locate the position of the first occurrence of substr column in the given string. |
| [**lcase**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.lcase.html#pyspark.sql.functions.lcase)(str) | Returns *str* with all characters changed to lowercase. |
| [**length**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.length.html#pyspark.sql.functions.length)(col) | Computes the character length of string data or number of bytes of binary data. |
| [**like**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.like.html#pyspark.sql.functions.like)(str, pattern[, escapeChar]) | Returns true if str matches *pattern* with *escape*, null if any arguments are null, false otherwise. |
| [**lower**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.lower.html#pyspark.sql.functions.lower)(col) | Converts a string expression to lower case. |
| [**left**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.left.html#pyspark.sql.functions.left)(str, len) | Returns the leftmost *len`(`len* can be string type) characters from the string *str*, if *len* is less or equal than 0 the result is an empty string. |
| [**levenshtein**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.levenshtein.html#pyspark.sql.functions.levenshtein)(left, right[, threshold]) | Computes the Levenshtein distance of the two given strings. |
| [**locate**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.locate.html#pyspark.sql.functions.locate)(substr, str[, pos]) | Locate the position of the first occurrence of substr in a string column, after position pos. |
| [**lpad**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.lpad.html#pyspark.sql.functions.lpad)(col, len, pad) | Left-pad the string column to width *len* with *pad*. |
| [**ltrim**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.ltrim.html#pyspark.sql.functions.ltrim)(col) | Trim the spaces from left end for the specified string value. |
| [**mask**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.mask.html#pyspark.sql.functions.mask)(col[, upperChar, lowerChar, digitChar, …]) | Masks the given string value. |
| [**octet\_length**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.octet_length.html#pyspark.sql.functions.octet_length)(col) | Calculates the byte length for the specified string column. |
| [**parse\_url**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.parse_url.html#pyspark.sql.functions.parse_url)(url, partToExtract[, key]) | Extracts a part from a URL. |
| [**position**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.position.html#pyspark.sql.functions.position)(substr, str[, start]) | Returns the position of the first occurrence of *substr* in *str* after position *start*. |
| [**printf**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.printf.html#pyspark.sql.functions.printf)(format, \*cols) | Formats the arguments in printf-style and returns the result as a string column. |
| [**rlike**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.rlike.html#pyspark.sql.functions.rlike)(str, regexp) | Returns true if *str* matches the Java regex *regexp*, or false otherwise. |
| [**regexp**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regexp.html#pyspark.sql.functions.regexp)(str, regexp) | Returns true if *str* matches the Java regex *regexp*, or false otherwise. |
| [**regexp\_like**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regexp_like.html#pyspark.sql.functions.regexp_like)(str, regexp) | Returns true if *str* matches the Java regex *regexp*, or false otherwise. |
| [**regexp\_count**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regexp_count.html#pyspark.sql.functions.regexp_count)(str, regexp) | Returns a count of the number of times that the Java regex pattern *regexp* is matched in the string *str*. |
| [**regexp\_extract**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regexp_extract.html#pyspark.sql.functions.regexp_extract)(str, pattern, idx) | Extract a specific group matched by the Java regex *regexp*, from the specified string column. |
| [**regexp\_extract\_all**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regexp_extract_all.html#pyspark.sql.functions.regexp_extract_all)(str, regexp[, idx]) | Extract all strings in the *str* that match the Java regex *regexp* and corresponding to the regex group index. |
| [**regexp\_replace**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regexp_replace.html#pyspark.sql.functions.regexp_replace)(string, pattern, replacement) | Replace all substrings of the specified string value that match regexp with replacement. |
| [**regexp\_substr**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regexp_substr.html#pyspark.sql.functions.regexp_substr)(str, regexp) | Returns the substring that matches the Java regex *regexp* within the string *str*. |
| [**regexp\_instr**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.regexp_instr.html#pyspark.sql.functions.regexp_instr)(str, regexp[, idx]) | Extract all strings in the *str* that match the Java regex *regexp* and corresponding to the regex group index. |
| [**replace**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.replace.html#pyspark.sql.functions.replace)(src, search[, replace]) | Replaces all occurrences of *search* with *replace*. |
| [**right**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.right.html#pyspark.sql.functions.right)(str, len) | Returns the rightmost *len`(`len* can be string type) characters from the string *str*, if *len* is less or equal than 0 the result is an empty string. |
| [**ucase**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.ucase.html#pyspark.sql.functions.ucase)(str) | Returns *str* with all characters changed to uppercase. |
| [**unbase64**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.unbase64.html#pyspark.sql.functions.unbase64)(col) | Decodes a BASE64 encoded string column and returns it as a binary column. |
| [**rpad**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.rpad.html#pyspark.sql.functions.rpad)(col, len, pad) | Right-pad the string column to width *len* with *pad*. |
| [**repeat**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.repeat.html#pyspark.sql.functions.repeat)(col, n) | Repeats a string column n times, and returns it as a new string column. |
| [**rtrim**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.rtrim.html#pyspark.sql.functions.rtrim)(col) | Trim the spaces from right end for the specified string value. |
| [**soundex**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.soundex.html#pyspark.sql.functions.soundex)(col) | Returns the SoundEx encoding for a string |
| [**split**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.split.html#pyspark.sql.functions.split)(str, pattern[, limit]) | Splits str around matches of the given pattern. |
| [**split\_part**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.split_part.html#pyspark.sql.functions.split_part)(src, delimiter, partNum) | Splits *str* by delimiter and return requested part of the split (1-based). |
| [**startswith**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.startswith.html#pyspark.sql.functions.startswith)(str, prefix) | Returns a boolean. |
| [**substr**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.substr.html#pyspark.sql.functions.substr)(str, pos[, len]) | Returns the substring of *str* that starts at *pos* and is of length *len*, or the slice of byte array that starts at *pos* and is of length *len*. |
| [**substring**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.substring.html#pyspark.sql.functions.substring)(str, pos, len) | Substring starts at *pos* and is of length *len* when str is String type or returns the slice of byte array that starts at *pos* in byte and is of length *len* when str is Binary type. |
| [**substring\_index**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.substring_index.html#pyspark.sql.functions.substring_index)(str, delim, count) | Returns the substring from string str before count occurrences of the delimiter delim. |
| [**overlay**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.overlay.html#pyspark.sql.functions.overlay)(src, replace, pos[, len]) | Overlay the specified portion of *src* with *replace*, starting from byte position *pos* of *src* and proceeding for *len* bytes. |
| [**sentences**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sentences.html#pyspark.sql.functions.sentences)(string[, language, country]) | Splits a string into arrays of sentences, where each sentence is an array of words. |
| [**to\_binary**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_binary.html#pyspark.sql.functions.to_binary)(col[, format]) | Converts the input *col* to a binary value based on the supplied *format*. |
| [**to\_char**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_char.html#pyspark.sql.functions.to_char)(col, format) | Convert *col* to a string based on the *format*. |
| [**to\_number**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_number.html#pyspark.sql.functions.to_number)(col, format) | Convert string ‘col’ to a number based on the string format ‘format’. |
| [**to\_varchar**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.to_varchar.html#pyspark.sql.functions.to_varchar)(col, format) | Convert *col* to a string based on the *format*. |
| [**translate**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.translate.html#pyspark.sql.functions.translate)(srcCol, matching, replace) | A function translate any character in the *srcCol* by a character in *matching*. |
| [**trim**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.trim.html#pyspark.sql.functions.trim)(col) | Trim the spaces from both ends for the specified string column. |
| [**upper**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.upper.html#pyspark.sql.functions.upper)(col) | Converts a string expression to upper case. |
| [**url\_decode**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.url_decode.html#pyspark.sql.functions.url_decode)(str) | Decodes a *str* in ‘application/x-www-form-urlencoded’ format using a specific encoding scheme. |
| [**url\_encode**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.url_encode.html#pyspark.sql.functions.url_encode)(str) | Translates a string into ‘application/x-www-form-urlencoded’ format using a specific encoding scheme. |

Bitwise Functions

|  |  |
| --- | --- |
| [**bit\_count**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bit_count.html#pyspark.sql.functions.bit_count)(col) | Returns the number of bits that are set in the argument expr as an unsigned 64-bit integer, or NULL if the argument is NULL. |
| [**bit\_get**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bit_get.html#pyspark.sql.functions.bit_get)(col, pos) | Returns the value of the bit (0 or 1) at the specified position. |
| [**getbit**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.getbit.html#pyspark.sql.functions.getbit)(col, pos) | Returns the value of the bit (0 or 1) at the specified position. |

Call Functions

|  |  |
| --- | --- |
| [**call\_function**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.call_function.html#pyspark.sql.functions.call_function)(funcName, \*cols) | Call a SQL function. |
| [**call\_udf**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.call_udf.html#pyspark.sql.functions.call_udf)(udfName, \*cols) | Call an user-defined function. |
| [**pandas\_udf**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.pandas_udf.html#pyspark.sql.functions.pandas_udf)([f, returnType, functionType]) | Creates a pandas user defined function (a.k.a. |
| [**udf**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.udf.html#pyspark.sql.functions.udf)([f, returnType, useArrow]) | Creates a user defined function (UDF). |
| [**udtf**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.udtf.html#pyspark.sql.functions.udtf)([cls, useArrow]) | Creates a user defined table function (UDTF). |
| [**unwrap\_udt**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.unwrap_udt.html#pyspark.sql.functions.unwrap_udt)(col) | Unwrap UDT data type column into its underlying type. |

Misc Functions

|  |  |
| --- | --- |
| [**aes\_decrypt**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.aes_decrypt.html#pyspark.sql.functions.aes_decrypt)(input, key[, mode, padding, aad]) | Returns a decrypted value of *input* using AES in *mode* with *padding*. |
| [**aes\_encrypt**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.aes_encrypt.html#pyspark.sql.functions.aes_encrypt)(input, key[, mode, padding, iv, aad]) | Returns an encrypted value of *input* using AES in given *mode* with the specified *padding*. |
| [**bitmap\_bit\_position**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bitmap_bit_position.html#pyspark.sql.functions.bitmap_bit_position)(col) | Returns the bit position for the given input column. |
| [**bitmap\_bucket\_number**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bitmap_bucket_number.html#pyspark.sql.functions.bitmap_bucket_number)(col) | Returns the bucket number for the given input column. |
| [**bitmap\_construct\_agg**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bitmap_construct_agg.html#pyspark.sql.functions.bitmap_construct_agg)(col) | Returns a bitmap with the positions of the bits set from all the values from the input column. |
| [**bitmap\_count**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bitmap_count.html#pyspark.sql.functions.bitmap_count)(col) | Returns the number of set bits in the input bitmap. |
| [**bitmap\_or\_agg**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.bitmap_or_agg.html#pyspark.sql.functions.bitmap_or_agg)(col) | Returns a bitmap that is the bitwise OR of all of the bitmaps from the input column. |
| [**current\_catalog**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.current_catalog.html#pyspark.sql.functions.current_catalog)() | Returns the current catalog. |
| [**current\_database**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.current_database.html#pyspark.sql.functions.current_database)() | Returns the current database. |
| [**current\_schema**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.current_schema.html#pyspark.sql.functions.current_schema)() | Returns the current database. |
| [**current\_user**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.current_user.html#pyspark.sql.functions.current_user)() | Returns the current database. |
| [**input\_file\_block\_length**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.input_file_block_length.html#pyspark.sql.functions.input_file_block_length)() | Returns the length of the block being read, or -1 if not available. |
| [**input\_file\_block\_start**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.input_file_block_start.html#pyspark.sql.functions.input_file_block_start)() | Returns the start offset of the block being read, or -1 if not available. |
| [**md5**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.md5.html#pyspark.sql.functions.md5)(col) | Calculates the MD5 digest and returns the value as a 32 character hex string. |
| [**sha**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sha.html#pyspark.sql.functions.sha)(col) | Returns a sha1 hash value as a hex string of the *col*. |
| [**sha1**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sha1.html#pyspark.sql.functions.sha1)(col) | Returns the hex string result of SHA-1. |
| [**sha2**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.sha2.html#pyspark.sql.functions.sha2)(col, numBits) | Returns the hex string result of SHA-2 family of hash functions (SHA-224, SHA-256, SHA-384, and SHA-512). |
| [**crc32**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.crc32.html#pyspark.sql.functions.crc32)(col) | Calculates the cyclic redundancy check value (CRC32) of a binary column and returns the value as a bigint. |
| [**hash**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.hash.html#pyspark.sql.functions.hash)(\*cols) | Calculates the hash code of given columns, and returns the result as an int column. |
| [**xxhash64**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.xxhash64.html#pyspark.sql.functions.xxhash64)(\*cols) | Calculates the hash code of given columns using the 64-bit variant of the xxHash algorithm, and returns the result as a long column. |
| [**assert\_true**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.assert_true.html#pyspark.sql.functions.assert_true)(col[, errMsg]) | Returns *null* if the input column is *true*; throws an exception with the provided error message otherwise. |
| [**raise\_error**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.raise_error.html#pyspark.sql.functions.raise_error)(errMsg) | Throws an exception with the provided error message. |
| [**reflect**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.reflect.html#pyspark.sql.functions.reflect)(\*cols) | Calls a method with reflection. |
| [**hll\_sketch\_estimate**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.hll_sketch_estimate.html#pyspark.sql.functions.hll_sketch_estimate)(col) | Returns the estimated number of unique values given the binary representation of a Datasketches HllSketch. |
| [**hll\_union**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.hll_union.html#pyspark.sql.functions.hll_union)(col1, col2[, allowDifferentLgConfigK]) | Merges two binary representations of Datasketches HllSketch objects, using a Datasketches Union object. |
| [**java\_method**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.java_method.html#pyspark.sql.functions.java_method)(\*cols) | Calls a method with reflection. |
| [**stack**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.stack.html#pyspark.sql.functions.stack)(\*cols) | Separates *col1*, …, *colk* into *n* rows. |
| [**try\_aes\_decrypt**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.try_aes_decrypt.html#pyspark.sql.functions.try_aes_decrypt)(input, key[, mode, padding, aad]) | This is a special version of *aes\_decrypt* that performs the same operation, but returns a NULL value instead of raising an error if the decryption cannot be performed. |
| [**typeof**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.typeof.html#pyspark.sql.functions.typeof)(col) | Return DDL-formatted type string for the data type of the input. |
| [**user**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.user.html#pyspark.sql.functions.user)() | Returns the current database. |
| [**version**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.version.html#pyspark.sql.functions.version)() | Returns the Spark version. |

Predicate Functions

|  |  |
| --- | --- |
| [**equal\_null**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.equal_null.html#pyspark.sql.functions.equal_null)(col1, col2) | Returns same result as the EQUAL(=) operator for non-null operands, but returns true if both are null, false if one of the them is null. |
| [**ifnull**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.ifnull.html#pyspark.sql.functions.ifnull)(col1, col2) | Returns *col2* if *col1* is null, or *col1* otherwise. |
| [**isnotnull**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.isnotnull.html#pyspark.sql.functions.isnotnull)(col) | Returns true if *col* is not null, or false otherwise. |
| [**nullif**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.nullif.html#pyspark.sql.functions.nullif)(col1, col2) | Returns null if *col1* equals to *col2*, or *col1* otherwise. |
| [**nvl**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.nvl.html#pyspark.sql.functions.nvl)(col1, col2) | Returns *col2* if *col1* is null, or *col1* otherwise. |
| [**nvl2**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.nvl2.html#pyspark.sql.functions.nvl2)(col1, col2, col3) | Returns *col2* if *col1* is not null, or *col3* otherwise. |

Xml Functions

|  |  |
| --- | --- |
| [**xpath**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.xpath.html#pyspark.sql.functions.xpath)(xml, path) | Returns a string array of values within the nodes of xml that match the XPath expression. |
| [**xpath\_boolean**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.xpath_boolean.html#pyspark.sql.functions.xpath_boolean)(xml, path) | Returns true if the XPath expression evaluates to true, or if a matching node is found. |
| [**xpath\_double**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.xpath_double.html#pyspark.sql.functions.xpath_double)(xml, path) | Returns a double value, the value zero if no match is found, or NaN if a match is found but the value is non-numeric. |
| [**xpath\_float**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.xpath_float.html#pyspark.sql.functions.xpath_float)(xml, path) | Returns a float value, the value zero if no match is found, or NaN if a match is found but the value is non-numeric. |
| [**xpath\_int**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.xpath_int.html#pyspark.sql.functions.xpath_int)(xml, path) | Returns an integer value, or the value zero if no match is found, or a match is found but the value is non-numeric. |
| [**xpath\_long**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.xpath_long.html#pyspark.sql.functions.xpath_long)(xml, path) | Returns a long integer value, or the value zero if no match is found, or a match is found but the value is non-numeric. |
| [**xpath\_number**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.xpath_number.html#pyspark.sql.functions.xpath_number)(xml, path) | Returns a double value, the value zero if no match is found, or NaN if a match is found but the value is non-numeric. |
| [**xpath\_short**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.xpath_short.html#pyspark.sql.functions.xpath_short)(xml, path) | Returns a short integer value, or the value zero if no match is found, or a match is found but the value is non-numeric. |
| [**xpath\_string**](https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.functions.xpath_string.html#pyspark.sql.functions.xpath_string)(xml, path) | Returns the text contents of the first xml node that matches the XPath expression. |

**when(condition\_on\_column, value)**

Evaluates a list of conditions and returns one of multiple possible result expressions.

If pyspark.sql.Column.otherwise() is not invoked, None is returned for unmatched conditions.

**Question:**

Create bellow dataframe and add one more column (paid\_status) in dataframe on condition and value-- When slary>125 then well\_pad else normal

|  |  |
| --- | --- |
| Source dataframe  +------+  |salary|  +------+  | 100|  | 110|  | 130|  | 105|  | 155|  +------+ | Resultant dataframe  +------+----------+  |salary|paid\_staus|  +------+----------+  | 100| normal|  | 110| normal|  | 130| well\_paid|  | 105| normal|  | 155| well\_paid|  +------+----------+ |

**Solution:**

from pyspark.sql.functions import \*

df=spark.createDataFrame([("100", ), ("110", ), ("130", ),("105", ),("155", )], ["salary"])

#Now use when function with otherwise

df.withColumn('paid\_staus',when(df.salary>125, 'well\_paid').otherwise('normal')).show()

**############################################################################**

**# Learned/Practice Question #**

**############################################################################**

data = [(1,),(2,),(3,)]

df1 = spark.createDataFrame(data, ["Column1"])

# Print DataFrame

df1.show()

data = [(11,1),(22,2),(33,3)]

df2 = spark.createDataFrame(data, ["Column2","Column1"])

# Print DataFrame

df2.show()

df1.join(df2,(df1.Column1==df2.Column1),how='inner').show()

**#Question#**

Create pyspark dataframe from multiple list. where each list contains a column data.

**Solution:**

list1 = ["a", "b", "c", "d"]

list2 = [1, 2, 3, 4]

df=spark.createDataFrame(tuple(zip(list1,list2)),schema='col1 string, col2 int')

**Note:**

createDataFrame function takes more data in list/tuple/iterable form for each row data.

<https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.sql.SparkSession.createDataFrame.html>

<https://saturncloud.io/blog/how-to-create-a-pyspark-dataframe-from-multiple-lists-a-comprehensive-guide/>

**Question:**

For below dataframe, replace name by NEW\_NAME string if gender if female (F)

+----+------+--------+

| id|c\_name|c\_gendre|

+----+------+--------+

| 1| AXEL| M|

| 2| Annie| F|

| 3| Ace| M|

| 4| Zelda| F|

| 5|Diesel| M|

| 11| Tilly| F|

| 7| Leroy| M|

|null|Olivia| F|

+----+------+--------+

**Solution**

from pyspark.sql.functions import when,regexp\_replace

df.withColumn('c\_name',when(df.c\_gendre=='M',regexp\_replace(df.c\_name,'^.\*$','NEW\_NAME') ).otherwise(df.c\_name) ).show()

**# PySpark Replace Column Values in DataFrame #**

We can check all details on below article-

<https://sparkbyexamples.com/pyspark/pyspark-replace-column-values/>

Let say we have below dataframe –

spark = SparkSession.builder.master("local[1]").appName("SparkByExamples.com").getOrCreate()

address = [(1,"14851 Jeffrey Rd","DE"),

    (2,"43421 Margarita St","NY"),

    (3,"13111 Siemon Ave","CA")]

df =spark.createDataFrame(address,["id","address","state"])

df.show()

+---+------------------+-----+

| id| address|state|

+---+------------------+-----+

| 1| 14851 Jeffrey Rd| DE|

| 2|43421 Margarita St| NY|

| 3| 13111 Siemon Ave| CA|

+---+------------------+-----+

**# PySpark Replace String Column Values #**

By using PySpark SQL function regexp\_replace() you can replace a column value with a string for another string/substring. regexp\_replace() uses Java regex for matching.

**Question**

For above dataframe relace Rd by Road in address column

**Solution**

from pyspark.sql.functions import regexp\_replace

df.withColumn('address', regexp\_replace('address', 'Rd', 'Road')) \

  .show(truncate=False)

**# Replace Column Values Conditionally #**

For doing conditional replacement we can use when().otherwise() SQL condition function.

**Example1** – already above, relace name column value based on gendre

**Example2**

Relace Rd by Road, St by Street and Av by Avenue in above dataframe.

**Solution:**

df.withColumn('address',

    when(df.address.endswith('Rd'),regexp\_replace(df.address,'Rd','Road')) \

   .when(df.address.endswith('St'),regexp\_replace(df.address,'St','Street')) \

   .when(df.address.endswith('Ave'),regexp\_replace(df.address,'Ave','Avenue')) \

   .otherwise(df.address)) \

   .show(truncate=False)

**# Replace Column Value with Dictionary (map) #**

This is done using, converting dataframe into rdd and then using map function.

Example:

stateDic={'CA':'California','NY':'New York','DE':'Delaware'}

df2=df.rdd.map(lambda x:

(x.id,x.address,stateDic[x.state])

).toDF(["id","address","state"])

df2.show()

**#Replace Column Value Character by Character#**

By using translate() string function you can replace character by character of DataFrame column value.

**Question:**

From above dataframe replace every character of 1 with A, 2 with B, and 3 with C on the address column.

**Solution:**

from pyspark.sql.functions import translate

df.withColumn('address', translate('address', '123', 'ABC')) \

  .show(truncate=False)

**Note:**

It can be done using regex\_replace() and replace() but translate() will be easy.

**#Question#**

Get the mean squared error for a dataframe which have real and predicted value for below dataframe

data = [(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]

df = spark.createDataFrame(data, ["actual", "predicted"])

**Solution**

df = df.withColumn("squared\_error", pow((col("actual") - col("predicted")), 2))

# Calculate the mean squared error

mse = df.agg({"squared\_error": "avg"}).collect()[0][0]

print(f"Mean Squared Error (MSE) = {mse}")

#Question#

Compute summary statistics(count, mean, stddev,min, max) for all columns in a dataframe.

Dataframe is-

data = [('James', 34, 55000),

('Michael', 30, 70000),

('Robert', 37, 60000),

('Maria', 29, 80000),

('Jen', 32, 65000)]

df = spark.createDataFrame(data, ["name", "age" , "salary"])

df.show()

Solution

print(type(df.describe()))

+-------+------+-----------------+-----------------+

|summary| name| age| salary|

+-------+------+-----------------+-----------------+

| count| 5| 5| 5|

| mean| null| 32.4| 66000.0|

| stddev| null|3.209361307176242|9617.692030835673|

| min| James| 29| 55000|

| max|Robert| 37| 80000|

+-------+------+-----------------+-----------------+

**#Question#**

convert year-month string to dates corresponding to the 4th day of the month for below dataframe

df = spark.createDataFrame([('Jan 2010',), ('Feb 2011',), ('Mar 2012',)], ['MonthYear'])

+---------+

|MonthYear|

+---------+

| Jan 2010|

| Feb 2011|

| Mar 2012|

+---------+

**Solution**

from pyspark.sql.functions import \*

df = spark.createDataFrame([('Jan 2010',), ('Feb 2011',), ('Mar 2012',)], ['MonthYear'])

#add string 04 to MonthYear column

df=df.withColumn('Date-Mmoth-Year',concat(lit('04').alias('Date-Mmoth-Year'),df.MonthYear))

#Now convert in into datetime ccolumn

df=df.withColumn('Date-Mmoth-Year',to\_date(col('Date-Mmoth-Year'),format='dMMM y'))

df.show()

df.dtypes

#**Question**#

From below email ids filter the valid email id.

data = ['buying books at amazom.com', 'rameses@egypt.com', 'matt@t.co', 'narendra@modi.com']

**Solution**:

Convert into dataframe then use rlike method.

data = ['buying books at amazom.com', 'rameses@egypt.com', 'matt@t.co', 'narendra@modi.com']

# Convert the list to DataFrame

#default column name will be value

df = spark.createDataFrame(data, "string")

#define the pattern for valid email id

pattern='[\w.-]+@[a-zA-Z]+\.[A-Za-z]+'

df.filter(col('value').rlike(pattern)).show()

#**Question**# ---- Good maths

Calculate the Euclidian distance between two series point.

Two series/dataframe is

data = [(1, 10), (2, 9), (3, 8), (4, 7), (5, 6), (6, 5), (7, 4), (8, 3), (9, 2), (10, 1)]

df = spark.createDataFrame(data, ["series1", "series2"])

+-------+-------+

|series1|series2|

+-------+-------+

| 1| 10|

| 2| 9|

| 3| 8|

| 4| 7|

| 5| 6|

| 6| 5|

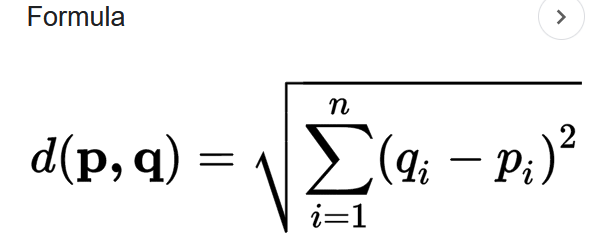
| 7| 4|

| 8| 3|

| 9| 2|

| 10| 1|

+-------+-------+



**#Solution#**

**Method1# ---- beginner approach**

Algo-

first calculate (p-q)\*\*2 for each element

then calculate the sum of (p-q)\*\*2

then calculate the square root of ( sum of( (p-q)\*\*2 ) )

#calcuate the (p-q)\*\*2

df=df.withColumn('result',pow((df.series1-df.series2),lit(2)))

#calculate the sqrt(sum of ( sum of (p-q)\*\*2 ) )

df.select(round(sqrt(sum(df.result)),2)).show()

**#Method 2# ---- Experts approach**

# Convert series to vectors

vecAssembler = VectorAssembler(inputCols=["series1", "series2"], outputCol="vectors")

df = vecAssembler.transform(df)

# Calculate squared differences

df = df.withColumn("squared\_diff", expr("POW(series1 - series2, 2)"))

# Sum squared differences and take square root

df.agg(expr("SQRT(SUM(squared\_diff))").alias("euclidean\_distance")).show()

#Question# --- try it, was unable to solve it

Replace missing spaces in a string with the least frequent character?

df = spark.createDataFrame([('dbc deb abed gade',),], ["string"])

df.show()

+-----------------+

| string|

+-----------------+

|dbc deb abed gade|

+-----------------+

Hint:

Use udf and Counter() from python collection.

Solution:

from pyspark.sql.functions import udf, explode

from pyspark.sql.types import StringType, ArrayType

from collections import Counter

def least\_freq\_char\_replace\_spaces(s):

counter = Counter(s.replace(" ", ""))

least\_freq\_char = min(counter, key = counter.get)

return s.replace(' ', least\_freq\_char)

udf\_least\_freq\_char\_replace\_spaces = udf(least\_freq\_char\_replace\_spaces, StringType())

df = spark.createDataFrame([('dbc deb abed gade',)], ["string"])

df.withColumn('modified\_string', udf\_least\_freq\_char\_replace\_spaces(df['string'])).show()

**#Question#** --- unable to solve it

Check if a dataframe has any missing values and count of missing values in each column?

**Solution**

from pyspark.sql.functions import col,isnan, when, count

df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df.columns]).show()

**#Question#**

From below dataframe get all rows for which value of **CID column is not '-'**

|  |  |
| --- | --- |
| Input  +----+----------+  | CID|trait\_diff|  +----+----------+  | a| a|  | a| null|  | -| a|  | -| null|  |null| a|  |null| null|  +----+----------+ | Exptd Output  +----+----------+  | CID|trait\_diff|  +----+----------+  | a| a|  | a| null|  |null| a|  |null| null|  +----+----------+ |

**Method1: --- this will not work ( because pyspark comparison is not null safe)**

df.filter(df.CID!='-').show()

**Output**

|  |  |
| --- | --- |
| +---+----------+  |CID|trait\_diff|  +---+----------+  | a| a|  | a| null|  +---+----------+ | **Explanation:**   * **In pyspark comparison operation is not nullSafe** * **If any of the operator is null then result of omparison will be null/unknow** * **In filter operation only those rows are returned whose comparison result is True**   <https://spark.apache.org/docs/3.5.4/sql-ref-null-semantics.html>  <https://medium.com/@pcbzmani/pyspark-null-safe-7e79306edca6>  OR  **Search for 'NULL SEMANTICS' on pyspark documentations** |

**Method 2: Correct answer**

df.registerTempTable("my\_temp\_table")

result = spark.sql("SELECT \* FROM my\_temp\_table where !(my\_temp\_table.CID<=>'-') ")

result.show()

**Method 2: Correct answer**

df=df.filter( ~(df.CID=='-') | (df.CID.isNull()) )

df.show()

**#Question 32#---** Good, unable to solve

For given dataframe find the number of null data in each column.

Dataframe is:

|  |  |
| --- | --- |
| df = spark.createDataFrame([  ("A", 1, None),  ("B", None, "123" ),  ("B", 3, "456"),  ("D", None, None),  ], ["Name", "Value", "id"])  df.show() | +----+-----+----+  |Name|Value| id|  +----+-----+----+  | A| 1|null|  | B| null| 123|  | B| 3| 456|  | D| null|null|  +----+-----+----+ |

#Method 1#

from pyspark.sql.functions import col,isnan, when, count

df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df.columns]   ).show()

**#Question 36#**

Format all the values in a dataframe as percentages. Dataframe is-

|  |  |
| --- | --- |
| Dataframe is:  data = [(0.1, .08), (0.2, .06), (0.33, .02)]  df = spark.createDataFrame(data, ["numbers\_1", "numbers\_2"])  df.show()  +---------+---------+  |numbers\_1|numbers\_2|  +---------+---------+  | 0.1 | 0.08|  | 0.2 | 0.06|  | 0.33 | 0.02|  +---------+---------+ |  |

Method 1:

from pyspark.sql.functions import concat, col, lit

cols=df.columns

for each in cols:

    df=df.withColumn(each+'\_percent',col(each)\*100)

df.show()

+---------+---------+-----------------+-----------------+

|numbers\_1|numbers\_2|numbers\_1\_percent|numbers\_2\_percent|

+---------+---------+-----------------+-----------------+

| 0.1| 0.08| 10.0| 8.0|

| 0.2| 0.06| 20.0| 6.0|

| 0.33| 0.02| 33.0| 2.0|

+---------+---------+-----------------+-----------------+

**#Question#** -- Assign sequential increasing ( deterministic number ) as index to dataframe.

**Solution**

df = df.withColumn("index", row\_number().over(Window.orderBy(monotonically\_increasing\_id()))-1 )

**#Question 37#** --- good, unable to solve

Get nth rows from given dataframe

+-------+------+

| Name|Number|

+-------+------+

| Alice| 1|

| Bob| 2|

|Charlie| 3|

| Dave| 4|

data = [("Alice", 1), ("Bob", 2), ("Charlie", 3), ("Dave", 4), ("Eve", 5),

("Frank", 6), ("Grace", 7), ("Hannah", 8), ("Igor", 9), ("Jack", 10)]

# Create DataFrame

df = spark.createDataFrame(data, ["Name", "Number"])

df.show()

**Hint:**

Use take(n) and check how take(n) differs from collect() --- method 1

Use row\_number and window function --- method 2

**Solution 1:**

#n= nth row want to get

print(df.take(n)[n-1])

**Solution2:** ---- By using row number

from pyspark.sql.functions import \*

from pyspark.sql.window import Window

#

window = Window.orderBy(monotonically\_increasing\_id())

# Add row\_number to DataFrame

df = df.withColumn("row\_number", row\_number().over(window))

#n=nth rows want to get

df.filter(df.row\_number==n).show()

**#Question #** ----- Good

Get every nth record from dataframe.

Example --- Get the every 5th row (rows for whose row number is multiple of 5) from pyspark dataframe

data = [("Alice", 1), ("Bob", 2), ("Charlie", 3), ("Dave", 4), ("Eve", 5),

("Frank", 6), ("Grace", 7), ("Hannah", 8), ("Igor", 9), ("Jack", 10)]

# Create DataFrame

df = spark.createDataFrame(data, ["Name", "Number"])

df.show()

|  |  |
| --- | --- |
| Input dataframe:  +-------+------+  | Name|Number|  +-------+------+  | Alice| 1|  | Bob| 2|  |Charlie| 3|  | Dave| 4|  | Eve| 5|  | Frank| 6|  | Grace| 7|  | Hannah| 8|  | Igor| 9|  | Jack| 10|  +-------+------+ | Expected result:  +----+------+  |Name|Number|  +----+------+  | Eve| 5|  |Jack| 10|  +----+------+ |

**Hint:**

Use row number and window function

**Solution:**

from pyspark.sql.functions import \*

from pyspark.sql.window import Window

#

window = Window.orderBy(monotonically\_increasing\_id())

# Add row\_number to DataFrame

df = df.withColumn("row\_number", row\_number().over(window))

#n=nth rows want to get

df.filter(df.row\_number%n==0).show()

#Question 38# --- good

Find the nth (n=integer) largest element from dataframe.

|  |  |
| --- | --- |
| from pyspark.sql import Row  # Sample Data  data = [  Row(id=1, column1=5),  Row(id=2, column1=8),  Row(id=3, column1=12),  Row(id=4, column1=1),  Row(id=5, column1=15),  Row(id=6, column1=7),  ]  df = spark.createDataFrame(data)  df.show() | +---+-------+  | id|column1|  +---+-------+  | 1| 5|  | 2| 8|  | 3| 12|  | 4| 1|  | 5| 15|  | 6| 7|  +---+-------+    1st largest element =15  3rd largest element will be --- 8 |

Hint:

Use window function.

**Solution**:

from pyspark.sql.window import Window

from pyspark.sql.functions import desc, row\_number

#

window=Window.orderBy(df.column1.desc())

df=df.withColumn('row\_num',row\_number().over(window))

df.show()

#**Question 39#** --- Good

Get the last n rows of dataframe where sum of each rows values <100.

# Sample data

data = [(10, 25, 70),(40, 5, 20),(70, 80, 100),(10, 2, 60),(40, 50, 20)]

# Create DataFrame

df = spark.createDataFrame(data, ["col1", "col2", "col3"])

# Display original DataFrame

df.show()

**Method 1: ----** beginner

from pyspark.sql import Row

df1=df.collect()

l=[]

row=Row()

for each in df1:

    row\_as\_dict=each.asDict()

    row\_sum=sum(list(row\_as\_dict.values()))

    l.append(Row(col1=row\_as\_dict['col1'],col2=row\_as\_dict['col2'],col3=row\_as\_dict['col3'],sum=row\_sum))

print(l)

#

df1 = spark.createDataFrame(l)

df1=df1.filter(df1.sum<100)

df1.show()

#Now write logic to get last n element

**Method 2 – Good for cconcept**

**get row-wise sum in new col, then create a index , sort in descending order of index**

from pyspark.sql import functions as F

from functools import reduce

# Add 'row\_sum' column

df = df.withColumn('row\_sum', reduce(lambda a, b: a+b, [F.col(x) for x in df.columns]))

# Display DataFrame with 'row\_sum'

df.show()

# Filter rows where 'row\_sum' > 100

df = df.filter(F.col('row\_sum') > 100)

# Display filtered DataFrame

df.show()

# Add 'id' column

df = df.withColumn('id', F.monotonically\_increasing\_id())\

# Get the last 2 rows

df\_last\_2 = df.sort(F.desc('id')).limit(2)

# Display the last 2 rows

df\_last\_2.show()

**Method3: ---** **Easiest** way

from pyspark.sql.functions import \*

# Sample data

data = [(10, 25, 70),(40, 5, 20),(70, 80, 100),(10, 2, 60),(40, 50, 20)]

# Create DataFrame

df = spark.createDataFrame(data, ["col1", "col2", "col3"])

# Display original DataFrame

df.show()

#get the row-wise sum of dataframe

df=df.withColumn('sum\_of\_cols',df.col1+df.col2+df.col3)

df.show()

#Filter the data for which sum<100

df=df.filter(df.sum\_of\_cols<100)

df.show()

#get the last n row; n=number

df=df.tail(1)

**#Question 40#** ----

**Calculate minimum by maximum of each row?**

data = [(1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12)]

# Create DataFrame

df = spark.createDataFrame(data, ["col1", "col2", "col3"])

df.show()

|  |  |
| --- | --- |
| Input Dataframe  +----+----+----+  |col1|col2|col3|  +----+----+----+  | 1| 2| 3|  | 4| 5| 6|  | 7| 8| 9|  | 10| 11| 12|  +----+----+----+ | Output Dataframe  +----+----+----+------------------+  |col1|col2|col3| min\_by\_max|  +----+----+----+------------------+  | 1| 2| 3|0.3333333333333333|  | 4| 5| 6|0.6666666666666666|  | 7| 8| 9|0.7777777777777778|  | 10| 11| 12|0.8333333333333334|  +----+----+----+------------------+ |

**Method 1:** ----- Using direct spark method

#get row wise minimum

df=df.withColumn('minimum',least(col('col1'),col('col2'),col('col3')))

#get row wise maximum

df=df.withColumn('maximum',greatest(col('col1'),col('col2'),col('col3')))

#calculate min/max

df=df.withColumn('min\_by\_max',df.minimum/df.maximum)

#select relevant col

df.select('col1','col2','col3','min\_by\_max').show()

Method 2: ---- Using udf

def min\_max\_ratio(row):

    row.sort()

    return row[0]/row[-1]

min\_max\_ratio\_udf = udf(min\_max\_ratio, FloatType())

# Apply UDF to create new column

df = df.withColumn('min\_by\_max', min\_max\_ratio\_udf(array(df.columns)))

df.show()

**Method3:**  ----- Using collect list

1. using array(cols) create a new column which will have array of each column data for each row.
2. Now create a result column to get max/min from above created column (created by array() )

**Example 40:** ---- How array(\*col) works

**arrary(\*col)**

It creates a column object for each row or df for columns ‘col’

Let say we have below dataframe –

+----+----+----+

|col1|col2|col3|

+----+----+----+

| 1| 2| 3|

| 4| 5| 6|

| 7| 8| 9|

| 10| 11| 12|

+----+----+----+

from pyspark.sql.functions import array

**df.select(array(df.columns)).collect()**

+-----------------------+

|array(col1, col2, col3)|

+-----------------------+

| [1, 2, 3]|

| [4, 5, 6]|

| [7, 8, 9]|

| [10, 11, 12]|

+-----------------------+

**#Example# ---- usage of array(\*col)**

For given dataframe, create a new col (each\_row\_data\_in\_list) which will contain the each row data in list form.

|  |  |
| --- | --- |
| **Input**  +----+----+----+  |col1|col2|col3|  +----+----+----+  | 1| 2| 3|  | 4| 5| 6|  | 7| 8| 9|  | 10| 11| 12|  +----+----+----+ | **Output:**  +----+----+----+----------------+  |col1|col2|col3|each\_row\_in\_list|  +----+----+----+----------------+  | 1| 2| 3| [1, 2, 3]|  | 4| 5| 6| [4, 5, 6]|  | 7| 8| 9| [7, 8, 9]|  | 10| 11| 12| [10, 11, 12]|  +----+----+----+----------------+ |

**Solution:**

from pyspark.sql import functions as F

data = [(1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12)]

# Create DataFrame

df = spark.createDataFrame(data, ["col1", "col2", "col3"])

df.show()

df=df.withColumn('each\_row\_in\_list',F.array(df.columns))

df.show()

**Questions:**

Create a new column in pyspark dataframe from list of values

|  |  |
| --- | --- |
| **Input dataframe**  +-------+---+--------------+-----+-----+  | name|age| subject|class| fees|  +-------+---+--------------+-----+-----+  | Arun| 16| Maths| 10|12000|  | Aniket| 17|Social Science| 11|15000|  | Ishita| 15| English| 9| 9000|  |Pranjal| 18| Science| 12|18000|  |Vinayak| 18| Computer| 12|18000|  +-------+---+--------------+-----+-----+ | Expected output  fine\_data = [200,300,400,0,500]  +-------+---+--------------+-----+-----+-----------+  | name|age| subject|class| fees|fine\_amt |  +-------+---+--------------+-----+-----+-----------+  | Arun| 16| Maths| 10|12000| 200|  | Aniket| 17|Social Science| 11|15000| 300|  | Ishita| 15| English| 9| 9000| 400|  |Pranjal| 18| Science| 12|18000| 0|  |Vinayak| 18| Computer| 12|18000| 500|  +-------+---+--------------+-----+-----+-----------+ |

Hint:

Create a index column in dataframe

Create a udf which will return a value from list for each value of index column data.

**Solution:**

from pyspark.sql.functions import \*

from pyspark.sql.types import IntegerType, StringType

file\_path=r'D:\Udemy\_PySpark\student\_data.csv'

df=spark.read.csv(file\_path,header=True)

df.show()

#list of data to assign as new column

fine\_data = [200,300,400,0,500]

#create a index column

df=df.withColumn('index',monotonically\_increasing\_id())

df.show()

#create udf, will retunr value from fine\_data list

@udf(returnType=IntegerType())

def my\_udf(x):

    return fine\_data[x]

df=df.withColumn('fine\_amount',my\_udf('index'))

df=df.select('name','age','subject','class','fees','fine\_amount')

df.show()

**#Question 41#**

Create a column that contains the **penultimate(second highest)** value in each row?

|  |  |
| --- | --- |
| **Input DataFrame:**  +-------+-------+-------+  |Column1|Column2|Column3|  +-------+-------+-------+  | 10| 20| 30|  | 40| 60| 50|  | 80| 70| 90|  +-------+-------+-------+ | **Output Dataframe:**  +-------+-------+-------+-----------+  |Column1|Column2|Column3|Penultimate|  +-------+-------+-------+-----------+  | 10| 20| 30| 20|  | 40| 60| 50| 50|  | 80| 70| 90| 80|  +-------+-------+-------+-----------+ |

**Hint**:

Use array and udf

**Solution**

from pyspark.sql import functions as F

from pyspark.sql.types import ArrayType, IntegerType

#

data = [(10, 20, 30),

(40, 60, 50),

(80, 70, 90)]

df = spark.createDataFrame(data, ["Column1", "Column2", "Column3"])

df.show()

# Define UDF to sort array in descending order

sort\_array\_desc = F.udf(lambda arr: sorted(arr), ArrayType(IntegerType()))

# Create array from columns, sort in descending order and get the penultimate value

df = df.withColumn("row\_as\_array", sort\_array\_desc(F.array(df.columns)))

df = df.withColumn("Penultimate", df['row\_as\_array'].getItem(1))

df = df.drop('row\_as\_array')

df.show()

**Note**:

For python array, there is not sort(), we have sorted(arr\_obj) for sorting.

**#Question 45#**

Get the frequency of unique values in the entire dataframe?

|  |  |
| --- | --- |
| **Input Dataframe**  +-------+-------+-------+  |Column1|Column2|Column3|  +-------+-------+-------+  | 1| 2| 3|  | 2| 3| 4|  | 1| 2| 3|  | 4| 5| 6|  | 2| 3| 4|  +-------+-------+-------+  data = [(1, 2, 3),  (2, 3, 4),  (1, 2, 3),  (4, 5, 6),  (2, 3, 4)]  df = spark.createDataFrame(data, ["Column1", "Column2", "Column3"])  # Print DataFrame  df.show() | **Expected Output**  +-------------+-----+  |single\_column|count|  +-------------+-----+  | 3| 4|  | 2| 4|  | 4| 3|  | 1| 2|  | 5| 1|  | 6| 1|  +-------------+-----+  Explanation:  3 have occurred 4 times in df  2 have occurred 4 times  5 have occurred 1 time |

**Method 1**: Beginner

1. Convert the df into list of Rows --- df.collect()
2. For each row get the values of each columns ( row.col\_name or row[col\_name] ) and append in another list

row\_list=df.collect()

l=[]

for row in row\_list:

    l.append('col1')

    l.append('col1')

    l.append('col3')

1. Now get the uniquq value of list and it’s occurance of each value

s=list(set(l))

cnt=[count l.count(each) for each in s]

1. Create new dataframe.

df = spark.createDataFrame(data, ["unique\_value", "value\_count"])

**Method** 2: ---- using union

# get column names

columns = df.columns

# stack all columns into a single column

df\_single = None

for c in columns:

    if df\_single is None:

        df\_single = df.select(col(c).alias("single\_column"))

    else:

        df\_single = df\_single.union(df.select(col(c).alias("single\_column")))

# generate frequency table

frequency\_table = df\_single.groupBy("single\_column").count().orderBy('count', ascending=False)

# show frequency table

frequency\_table.show()

**#Example# --- concatenating/placing two dataframe**

Perform concatenating/placing two dataframe side by side

**Hint**:

Assign index column in both dataframe

Now perform inner join on index column

**#Question 47#**

Reverse the column data of a dataframe.

|  |  |
| --- | --- |
| **Input Dataframe**  +-----+-----+-----+-----+  |col\_1|col\_2|col\_3|col\_4|  +-----+-----+-----+-----+  | 1| 2| 3| 4|  | 2| 3| 4| 5|  | 3| 4| 5| 6|  | 4| 5| 6| 7|  +-----+-----+-----+-----+ | **Expected Output**  +-----+-----+-----+-----+  |col\_1|col\_2|col\_3|col\_4|  +-----+-----+-----+-----+  | 4| 5| 6| 7|  | 3| 4| 5| 6|  | 2| 3| 4| 5|  | 1| 2| 3| 4|  +-----+-----+-----+-----+ |

**Method1:** --- beginner approach

For each column do below operation:-

1. First convert the that column data into list, using some sort of below code.

for each in df.columns:

    print(type(df.select(each).rdd))

    rdd=df.select(each).rdd

    rdd=rdd.map( lambda x : x[each])

    l=rdd.collect()

1. Reverse the list data using – l=l[::-1]
2. Now using a udf function assign the reversed list data to same column.

**Method 2: ---- Experts way**

Create a index column

Sort the df in descending order of index

Delete the index column at end

from pyspark.sql.window import Window

from pyspark.sql.functions import row\_number, monotonically\_increasing\_id

# Define window specification

w = Window.orderBy(monotonically\_increasing\_id())

# Add index

df = df.withColumn("id", row\_number().over(w) - 1)

# Sort the dataframe

df\_2 = df.orderBy("id", ascending=False).drop("id")

df\_2.show()

**# Example #** **---- forward\_fill concept**

<https://medium.com/@aman.gupta435433/performing-backward-filling-and-forward-filling-operations-in-pyspark-2065aafa9ff0>

Let say we have one dataframe where one column 'date' data is missing in between.

Perform forward fill operation.

|  |  |
| --- | --- |
| Input Dataframe  +---------+----------------+---------+  | Date|Completed Trips |Requests |  +---------+----------------+---------+  |10-Sep-12| 2| 2|  | null| 2| 2|  | null| 0| 0|  |11-Sep-12| 1| 1|  | null| 0| 1|  | null| 0| 0|  |12-Sep-12| 0| 1|  | null| 1| 1|  +---------+----------------+---------+ | Expeacted/Output Dataframe  +---------+----------------+---------+  | Date|Completed Trips |Requests |  +---------+----------------+---------+  |10-Sep-12| 2| 2|  |10-Sep-12| 2| 2|  |10-Sep-12| 0| 0|  |11-Sep-12| 1| 1|  |11-Sep-12| 0| 1|  |11-Sep-12| 0| 0|  |12-Sep-12| 0| 1|  |12-Sep-12| 1| 1|  +---------+----------------+---------+ |

**Hint**:

Create the forward moving frame

Fill the last value with current value using last()

**Solution**:

#Create the forward frame window

window = Window.rowsBetween(float('-inf'),0)

#Now using last method do forward fill oepartion

filled\_column\_1 = last(df['Date'], ignorenulls=True).over(window)

#repalce the column values

df=df.withColumn('Date',filled\_column\_1)

print(df.show(20))

**# Example #** **---- backward\_fill concept**

<https://medium.com/@aman.gupta435433/performing-backward-filling-and-forward-filling-operations-in-pyspark-2065aafa9ff0>

Let say we have below dataframe and want to do backward fill operation for column ‘a’.

|  |  |
| --- | --- |
| Input Dataframe  +----+----+----+----+----+ | a| b| c| d| e| +----+----+----+----+----+ | 1| 8| abc| 14| 4| | 2|null| def|null|null| |null|null|null|null| 9| |null| 9|null|null| 9| | 7| 9|null| 18|null| | 8| 9| xyz| 18|null| +----+----+----+----+----+ | Expected/Output dataframe  +---+----+----+----+----+  | a| b| c| d| e|  +---+----+----+----+----+  | 1| 8| abc| 14| 4|  | 2|null| def|null|null|  | 7|null|null|null| 9|  | 7| 9|null|null| 9|  | 7| 9|null| 18|null|  | 8| 9| xyz| 18|null|  +---+----+----+----+----+ |

**Hint**:

Create the backward moving frame

Fill the current value with last value using first()

**Solution:**

## specifying direction of filling

window = Window.rowsBetween(0,float('inf'))

# defining the backward-filled column

filled\_column\_1 = first(df['a'], ignorenulls=True).over(window)

df.withColumn('a', filled\_column\_1).show(50)

**#Example**# **window /frame size and direction in window ---------- Good**

<https://stackoverflow.com/questions/70771993/window-function-acts-not-as-expected-when-i-use-order-by-pyspark>

Let say we have below dataframe, analyse why is the different in output in Method1 and Metho2.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input dataframe**   |  |  | | --- | --- | | CATEGORY | REVENUE | | Cell Phone | 6000 | | Tablet | 1500 | | Tablet | 5500 | | Cell Phone | 5000 | | Cell Phone | 6000 | | Tablet | 2500 | | Cell Phone | 3000 | | Cell Phone | 3000 | | Tablet | 3000 | | Tablet | 4500 | | Tablet | 6500 | | columns = ["CATEGORY", "REVENUE"]  data = [("Cell Phone", "6000"),("Tablet", "1500"),("Tablet", "5500"),("Cell Phone", "5000"),("Cell Phone", "6000"),  ("Tablet", "2500"),("Cell Phone", "3000"),("Cell Phone", "3000"),("Tablet", "3000"),("Tablet", "4500"),  ("Tablet", "6500")]  df = spark.createDataFrame(data=data, schema=columns)  df.show() |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Method 1:**  window\_spec = Window.partitionBy(df['CATEGORY'])  #  df=df.withColumn('max\_revenue',  F.max(df['REVENUE']).over(window\_spec))   |  |  |  | | --- | --- | --- | | | CATEGORY|R | EVENUE | max(revenue) | | |Cell Phone| | 6000| | 6000| | | |Cell Phone| | 5000| | 6000| | | |Cell Phone| | 6000| | 6000| | | |Cell Phone| | 3000| | 6000| | | |Cell Phone| | 3000| | 6000| | | | Tablet| | 1500| | 6500| | | | Tablet| | 5500| | 6500| | | | Tablet| | 2500| | 6500| | | | Tablet| | 3000| | 6500| | | | Tablet| | 4500| | 6500| | | | Tablet| | 6500| | 6500| | | Method 2:  window\_spec = Window.partitionBy(df['CATEGORY']).orderBy("REVENUE")  #  df=df.withColumn('max\_revenue',F.max(df['REVENUE']).over(window\_spec))   |  |  |  | | --- | --- | --- | | CATEGORY | REVENUE | max\_revenue | | Cell Phone | 3000 | 3000 | | Cell Phone | 3000 | 3000 | | Cell Phone | 5000 | 5000 | | Cell Phone | 6000 | 6000 | | Cell Phone | 6000 | 6000 | | Tablet | 1500 | 1500 | | Tablet | 2500 | 2500 | | Tablet | 3000 | 3000 | | Tablet | 4500 | 4500 | | Tablet | 5500 | 5500 | | Tablet | 6500 | 6500 | |

**Explanation:**

* when we add orderBy to the Windows definition, Spark will maintain a logical pointer within each partition. As the pointer moves through each row, the window function is computed based on the ordered rows up to that point, considering the defined frame. This would be handy for cases like ranking or numbering the rows within a partition
* when the Window is an unordered one, Spark will consider the window function over the entire partition at once. The order of rows within the partition is not explicitly defined, so the window function will operate on all rows in the partition as a single frame.
* We **can fix the problem of method by using rowsBetween() which will enlarge the frame** size ( from **unboundedPreceding, unboundedFollowing )** as below-

Using unboundedPreceding (last pointer) and unboundedFollowing(current/next pointer) -

**Window.partitionBy(df['CATEGORY']).orderBy(df['REVENUE']).rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)**

**#Question** # Difference between rowsBetween() and rangeBetween()

<https://www.youtube.com/watch?v=4eEIs9gtKjw>

**# Example #** ----- **Replace the null by average value**.

Let say we have below dataframe and want to replace the null/missing value of salary column by average of salary.

+---------+----------+--------+-----+------+------+

|firstname|middlename|lastname|id |gender|salary|

+---------+----------+--------+-----+------+------+

|James | |Smith |36636|M |3000 |

|Michael |Rose | |40288|M |4000 |

|Robert | |Williams|42114|M |4000 |

|Maria |Anne |Jones |39192|F |4000 |

|Jen |Mary |Brown | |F |null |

+---------+----------+--------+-----+------+------+

**Method 1: ----- Without using inbuild method**

Hint:

* First the salary column data in another dataframe
* Using agg function get the salary then ---- > conver into dict

then ----- > withColumn method replace the null/missing value by average value.

#gett eh salary into new dataframe

df1=df.select('salary').filter(df.salary>=0)

#convert the salary data into dict as {'col\_name':"average\_of\_col"}

my\_dict=df1.agg(avg('salary').alias('avg\_salary')).first().asDisct()

#Now replcae the missing value by average value

df=df.fillna('salary',my\_dict['avg\_salary'])

Method 2: ---- Using Impute from pysapk.

**#Example: -- dataframe transform**

Let say we have below dataframe, now create new column for by applying below transformation.

new\_fee = feee-100

discount\_fee = new\_fee-(-(new\_fee\*discount)/100)

|  |  |
| --- | --- |
| **Input dataframe**  +----------+----+--------+  |CourseName|fee |discount|  +----------+----+--------+  |Java |4000|5 |  |Python |4600|10 |  |Scala |4100|15 |  |Scala |4500|15 |  |PHP |3000|20 |  +----------+----+--------+ | **Output dataframe**  +----------+----+--------+-------+--------------+  |CourseName| fee|discount|new\_fee|discounted\_fee|  +----------+----+--------+-------+--------------+  | JAVA|4000 | 5 | 3000| 2850.0|  | PYTHON|4600| 10 | 3600| 3240.0|  | SCALA|4100| 15 | 3100| 2635.0|  | SCALA|4500| 15 | 3500| 2975.0|  | PHP|3000| 20 | 2000| 1600.0|  +----------+----+--------+-------+--------------+ |

simpleData = (("Java",4000,5), ("Python", 4600,10),("Scala", 4100,15), ("Scala", 4500,15), ("PHP", 3000,20), ),columns= ["CourseName", "fee", "discount"]

# Create DataFrame

df = spark.createDataFrame(data = simpleData, schema = columns)

**Solution:**

# Custom transformation 1

from pyspark.sql.functions import upper

# Custom transformation 2, passing some value while calling function

def reduce\_price(df,**reduceBy**): # **reduceBy for receiving the passed value**

    return df.withColumn("new\_fee",df.fee - reduceBy)

# Custom transformation 3, passing now value

def apply\_discount(df):

    return df.withColumn("discounted\_fee",df.new\_fee - (df.new\_fee \* df.discount) / 100)

# PySpark transform() Usage

df2 = df.transform(reduce\_price,1000).transform(apply\_discount)

df2.show()

**Note**:

Here we can see that we are not passing then column to function but calling function of dataframe which passed the whole dataframe to called function.

**#Example: -- functions transform**

The PySpark sql.functions.transform() is used to apply the transformation on a column of type Array.

This function applies the specified transformation on every element of the array and returns an object of ArrayType.

pyspark.sql.functions.transform( col, func ) --- syntax

**Question: -- Change the column data( assuming column data is arrary type ) into upper case using pyspark,sql.functions.transform()**

|  |  |
| --- | --- |
| Input dataframe  +-------------------------+---------------------------+  | Name | Languages1|  +---------------- --------+---------------------------+  | James,,Smith | [Java, Scala, C++] |  | Michael,Rose, | [Spark, Java, C++] |  |Robert,,Williams | [CSharp, VB] |  +-------------------------+---------------------------+ | Expected output dataframe  +-------------------------+---------------------------+  | Name | Languages1|  +---------------- --------+---------------------------+  | James,,Smith | [JAVA, SCALA, C++] |  | Michael,Rose, | [SPARK, JAVA, C++] |  |Robert,,Williams | [CSHARP, VB] |  +-------------------------+---------------------------+ |

data = [

("James,,Smith",["Java","Scala","C++"]),

("Michael,Rose,",["Spark","Java","C++"]),

("Robert,,Williams",["CSharp","VB"])

]

df = spark.createDataFrame(data=data,schema=["Name","Languages1"])

**# Example on pivot #**

**GroupedData.pivot(pivot\_col: str, values: Optional[List[LiteralType]] = None)**

**GroupedData.pivot(pivot\_col: str, values: Optional[List[LiteralType]] = None).agg(func1,fun2…)**

* **pivot\_col** --- Name of col to pivot

**this col value(distinct) will be created as column name in output df**

* **values** --- List of values that will be translated to column in output dataframe

**value from pivot\_col to use for creating col in result df, default – all values.**

Pivoting is a data transformation technique that involves converting rows into columns. This operation is valuable when reorganizing data for enhanced readability, aggregation, or analysis.

**Note**:

Output dataframe will always have - grouping col, pivoted col (or values around which pivot is performed <value> parameter in pivot ) and other col on which any operation is done(if doing aggregation on more than one col.) --- Check Example – pivot of cricket

**Question**:

Compute the sum of earnings for each year by course with each course as a separate column

|  |  |
| --- | --- |
| **Input dataframe**  +-------------+----+------------+  |course |year|earnings|  +------------+----+-------------+  |dotNET |2012| 10000|  | Java |2012| 20000|  |dotNET |2012| 5000 |  |dotNET |2013| 48000|  | Java |2013| 30000|  +------------+-----+-----------+ | **Expected dataframe**  +-----------+----------+---------+  |year |dotNET| Java |  +-----------+----------+---------+  |2012 | 15000 |20000 |  |2013 | 48000 |30000 |  +-----------+-----------+--------+ |

from pyspark.sql import Row

df1 = spark.createDataFrame([ Row(course="dotNET", year=2012, earnings=10000),Row(course="Java", year=2012, earnings=20000),

Row(course="dotNET", year=2012, earnings=5000),Row(course="dotNET", year=2013, earnings=48000),Row(course="Java", year=2013, earnings=30000),

])

**Solution**:

df1.groupBy("year").pivot("course").sum("earnings").show()

OR ---- by specifying the column along with col to pivot

df1.groupBy("year").pivot("course", ["dotNET", "Java"]).sum("earnings").show()

Question

Compute the sum of earnings for each year by course with each course as a separate column.

Sales column have course\_name, ear and earning data in array form.

|  |  |
| --- | --- |
| **Input dataframe**  +--------------+---------------------------+  |training | sales |  +-------------+----------------------------+  | expert |{dotNET, 2012, 1000} |  | junior | {Java, 2012, 20000} |  | expert |{dotNET, 2012, 5000}|  | junior |{dotNET, 2013, 4800} |  | expert | {Java, 2013, 30000} |  +----------------+-------------------------+ | **Expected output**  +----+-----+------+  |year| Java|dotNET|  +----+-----+------+  |2012|20000| 15000|  |2013|30000| 48000|  +----+-----+------+ |

df2 = spark.createDataFrame([

Row(training="expert", sales=Row(course="dotNET", year=2012, earnings=10000)),

Row(training="junior", sales=Row(course="Java", year=2012, earnings=20000)),

Row(training="expert", sales=Row(course="dotNET", year=2012, earnings=5000)),

Row(training="junior", sales=Row(course="dotNET", year=2013, earnings=48000)),

Row(training="expert", sales=Row(course="Java", year=2013, earnings=30000)),

])

**Solution**

df2.groupBy("sales.year").pivot("sales.course").sum("sales.earnings").show()

#################

# PIVOTING/PIVOT #

#################

**# Example # --- Pivot on cricket DF**

Let say we have below dataframe,

****

match\_data = [

("Jadeja", 'Wankhede', 40, 2, 1), ("Hardik", 'Eden Gardens', 55, 0, 2),

("Jadeja", 'Eden Gardens', 25, 3, 0), ("Watson", 'Wankhede', 30, 1, 0),

("Hardik", 'Wankhede', 45, 0, 1), ("Jadeja", 'Wankhede', 60, 1, 1),

("Hardik", 'Eden Gardens', 45, 0, 3), ("Hardik", 'Eden Gardens', 30, 4, 0),

("Watson", 'Wankhede', 20, 2, 2), ("Watson", 'Eden Gardens', 50, 0, 0)

]

match\_schema = "Player\_Name String , Stadium String , Runs int , Wickets int , Catches int"

match\_df = spark.createDataFrame(data = match\_data, schema = match\_schema)

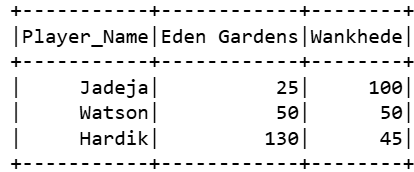
**Questions -**

1. Run scored by each player in each stadium, by creating each stadium name as col name
2. Calculate the run scored, wicket taken in each stadium, by creating each stadium name as col name

**Question a**

Run scored by each player in each stadium, by creating each stadium name as col name

Expected output --



**Solution**:

from pyspark.sql.functions import sum

match\_df.groupBy('Player\_Name').pivot('Stadium').agg(sum('Runs')).show()

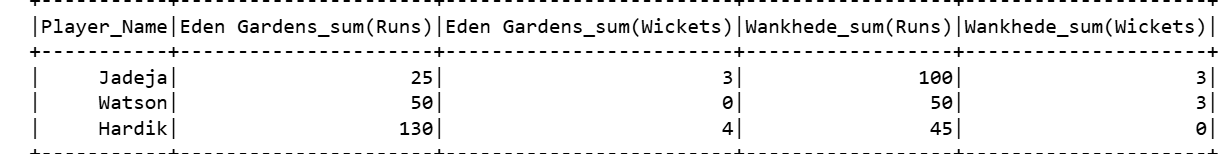
**Note:**

**Only one aggregation on pivoted data is applied so in output dataframe don't have extra column as expected ( as mention in note section above)**

**Question b**

Calculate the run scored, wicket taken in each stadium, by creating each stadium name as col name

Expected result



**Solution**:

from pyspark.sql.functions import sum

match\_df.groupBy('Player\_Name').pivot('Stadium').agg(sum('Runs'),sum('Wickets')).show()

**Note**:

**Here we can see it have created new column for each stadium for runs and wicket as expected because applying more than one aggregation function ( as mention in Note section above)**

**#########**

**#unpivot #**

**#########**

Unpivot is changing from wide format to long format (col to row operation) i.e reverse of pivot.

In pyspark there is no method by for unpivot operation but we can archive it using stack() and expr()

#############

#**Example expr#**

**##############**

pyspark.sql.functions.expr(str: str)

Returns the pyspark column object after the evaluating the expression.

This evaluates the given expression cell-wise. (Check question – " **Question – expr\_3**" in this sheet)

Parameter:

* str : str, expression/SQL expression or other expression/operation defined in string.

**Example** : **concatenating first name and last name column separated by comma(,)**

Let say we have df with first\_name and last\_name column. Concatenate the first\_name and last\_name.

|  |  |
| --- | --- |
| Input dataframe | Expected output |

**Solution**:

df.withColumn("Name", expr( " col1 || ',' || col2" ) ).show()

**Note:**

Here we have concatenated col1(first\_name) with comm(,) then col2 (last name)

**Question** : **expr with case and when statement** (can be without expr() but do with expr() )

Let say we have below dataframe , now we have to transform the dataframe gender column based on rule – when gender=F then assign the value Female and when gender=M then assign Male.

|  |  |
| --- | --- |
| Input dataframe    from pyspark.sql.functions import expr  data = [("James","M"),("Michael","F"),("Jen","")]  columns = ["name","gender"]  df = spark.createDataFrame(data = data, schema = columns) | Expected output |

Solution:

df2=df.withColumn("gender", expr("CASE WHEN gender = 'M' THEN 'Male' " +

"WHEN gender = 'F' THEN 'Female' ELSE 'unknown' END" ) )

**#Question – expr\_3#** apply transformation given in one column to other column ------- **Good**

Let say we have one dataframe, in which we have which column which shows the transformation rule to be applied for another column data for each value.

Apply the transformation and create a new column.

|  |  |
| --- | --- |
| Input Dataframe | Expected ouput |

**Hint:**

Use expr function

**Solution:**

from pyspark.sql.functions import \*

for row in df\_with\_regex.collect():

    df\_with\_regex = df\_with\_regex.withColumn('transform\_data',expr(row['expr\_to\_apply']))

df\_with\_regex.show(truncate=False)

**# Example # --- unpivot datarfame**

In pyspark **there is no method for unpivoting** dataframe but we can achieve it by using expr() and stack().

stack(n, (' value\_for\_col1 ',col1\_of\_df, ' value\_for\_col2',col2\_of\_df, .. ), as (Col\_Name, result\_col) ) --------- Stack Syntax

* Col\_Name this is the column in result df , showing from which col of input dataframe values are placed in result\_col column.

expr(str: str) --------- syntax of expr, check above for xample

**#Question# ---- unpivot dataframe**

Let say we have below dataframe, now do unpivot to get the expected output dataframe.

This dataframe contains the run score by each player in each match

|  |  |
| --- | --- |
| **Input dataframe**    cricket\_data = [("Virat Kohli", 85, 100, 75),  ("Steve Smith", 90, 105, 80),  ("Kane Williamson", 88, 95, 70)]  df = spark.createDataFrame(cricket\_data, ["Player", "Match1", "Match2", "Match3"])  df.show() | Output dataframe |

**Solution**

from pyspark.sql.functions import expr

stack\_expr="stack(3,'Match1', Match1,'Match2', Match2,'Match3',Match3) as (Match, score)"

df.select('Player',expr(stack\_expr)).show()

**#Example# ---- unpivoting**

Let way we have below dataframe and **want to unpivot taking (Dimension1, and Dimension2) as index/id column to unpivot the Hierarchy1, Hierarchy2, Hierarchy3 column.**

|  |  |
| --- | --- |
| **Input dataframe**    df = spark.createDataFrame([ ("D1", "D2", "H1", None, None),("D1", "D2", "H1", "H2", None),("D1", "D2", "H1", "H2", "H3"),  ("D2", "D2", "H1", "H5", "H3")],  schema=["Dimension1", "Dimension2", "Hierarchy1", "Hierarchy2", "Hierarchy3"] )  df.show() | **Expected output** |

**Hint**:

Use Dimension1 and Dimension2 as index/id variable.

In stack use Hierarchy1, Hierarchy2, Hierarchy3 and in as (col\_name, res\_col) given any name.

**Solution**:

# Stack and explode the columns to get distinct hierarchy values

stack\_expr="stack(3, 'Hierarchy1', Hierarchy1, 'Hierarchy2', Hierarchy2, 'Hierarchy3', Hierarchy3) as **(Hierarchy\_Col\_Name, Hierarchy)"**

stacked\_df = df.selectExpr("Dimension1","Dimension2",stack\_expr)

# Show the resulting DataFrame

stacked\_df.show()

**#Question#** ---- unpivoting/melting using panda

Do the above unpivot/melt operation using pandas library.

**Solution:**

import pandas  as pd

pd.melt(pd\_df,id\_vars=['Dimension1','Dimention2'],var\_name='Hierarchy\_Col\_Name',value\_name='Hierarchy')

**#Example # --- vertical stacking of dataframe**

For doing the vertical stacking of dataframe we can use pyspark unionByName() function

**df1.unionByName(df2, allowMissingColumns = bool)** -------- Syntax

**Parameter**

**allowMissingColumnsbool=Bool**, Specify whether to allow missing columns

If passed True then uncommon column will come in result filled by None

If passed False then only common column will come in result

**Question**: **vertical stacking / unionByName (allowMissingColumns=True)**

Stack the below two dataframe column wise/vertical stacking.

|  |  |
| --- | --- |
| Input dataframe      columns=['Name','age']  data=[('Shyam',24),('Chiku',30)]  df1=spark.createDataFrame(data=data,schema=columns)  df1.show()  #  columns=['Age','Salary']  data=[(24,400),(30,5400)]  df2=spark.createDataFrame(data=data,schema=columns)  df2.show() | Expected Output |

**Solution**:

df1.unionByName(df2,allowMissingColumns=True).show()

**#Question # --- stackoverflow ---- creating category based on column data.**

Let say we have one dataframe (product ) having 5 column, now based on the type of each column data classify them in separate category which will be integer type.

|  |  |
| --- | --- |
| Input dataframe    df=spark.read.format("jdbc").options(  url="jdbc:mysql://localhost:3306/learn",dbtable = "products",user="root",password="admin").load() | Output dataframe |

**Hint:**

* Create a new column, which will contains all unique value of each row in a list/array
* Now on new\_created column use dense\_rank function

**Solution**:

#Now remove the null values from arr\_col

from pyspark.sql.functions import \*

#create the column arr\_col,

# It will contain each row data for all column in list/array type

df=df.withColumn('arr\_col',array('node\_0','node\_1','node\_2','node\_3','node\_4'))

#Now remove the null value from arr\_col column

#remove of null can be done by UDF also

df1=df.withColumn("node\_array\_clean", F.filter(F.col("arr\_col"), lambda x: x.isNotNull()))

df1=df1.select('node\_0','node\_1','node\_2','node\_3','node\_4','node\_array\_clean')

#Now use Window function and dense\_rank

window\_spec = Window.partitionBy().orderBy("node\_array\_clean")

df1=df1.withColumn('catagerory',dense\_rank().over(window\_spec))

df1=df1.select('node\_0','node\_1','node\_2','node\_3','node\_4','catagerory')

df1.show()

**#Question #** -- **find column name having second max value for each row** ----- StackoverFlow

<https://stackoverflow.com/questions/78708786/how-to-get-the-name-of-column-with-second-highest-value-in-pyspark-dataframe>

Let say we have transaction data in pyspark dataframe having column name ( 'trx\_holiday', 'trx\_takeout', 'trx\_pet', 'max\_value' ).

Create the column MAX\_COL\_NM\_FOR\_EACH\_ROW which will hold the column name in each row which have maximum transaction amount.

|  |  |
| --- | --- |
| Input data    from pyspark.sql.functions import \*  data=[(12.5,5.5,9.5,12.5),(3.0,14.0,6.7,14.0)]  schema=('trx\_holiday','trx\_takeout',  'trx\_pet','max\_value')  df=spark.createDataFrame(data=data,  schema=schema)  df.printSchema()  df.show() | Expected output |

**Hint:**

Use unpivot , window (dense rank) on unpivoted dd and then join

from pyspark.sql.functions import \*

from pyspark.sql import Window

data=[(12.5,5.5,9.5,12.5),(3.0,14.0,6.7,14.0)]

schema=('trx\_holiday','trx\_takeout','trx\_pet','max\_value')

df=spark.createDataFrame(data=data,schema=schema)

df.show()

#create the index/id/pkey column

df=df.withColumn('id',monotonically\_increasing\_id())

#unpivot/melt the dataframe

stack\_expr="stack(4,'trx\_holiday',trx\_holiday,'trx\_takeout',trx\_takeout,'trx\_pet',trx\_pet,'max\_value',max\_value) as (txn\_type,amount)"

df1=df.selectExpr("id",stack\_expr)

#apply window

window\_spec=Window.partitionBy('id').orderBy(desc('amount'))

df1=df1.withColumn('dense\_rank',dense\_rank().over(window\_spec))

df1=df1.filter(df1.dense\_rank==2).select('id','txn\_type')

#Now join the df1 with df

df=df.join(df1,on=(df.id==df1.id),how='left').withColumnRenamed('txn\_type','MAX\_COL\_NM\_FOR\_EACH\_ROW')

df.show()

**#Question #** ---- creating empty dataframe

schema=<create pyspark schema>

schema = StructType([StructField("A", IntegerType(), True),StructField("B", IntegerType(), True),])

empty\_df = spark.createDataFrame(spark.sparkContext.emptyRDD(), schema)

**#Example # Assigning/creation a dict\_value to pyspark column ------ Good, Concept**

Let say we have pyspark dataframe, now in this dataframe, we want to assign/create new column which hold dict type value.

|  |  |
| --- | --- |
| **Input DataFrame**  +---+  | id|  +---+  | 1|  | 2|  +---+  df = spark.createDataFrame([(1,),(2,)],  "id int") | **Output DataFrame**  +---+--------------------------------------------+  |id |cam\_details |  +---+--------------------------------------------+  |1 |{1234, [6784, 8901], 2002-05-06}|  |2 |{1234, [6784, 8901], 2002-05-06}|  +---+--------------------------------------------+  **Schema detail:**  root  |-- id: integer (nullable = true)  |-- cam\_details: struct (nullable = true)  | |-- emp\_id: integer (nullable = true)  | |-- emp\_acct: array (nullable = true)  | | |-- element: integer (containsNull = true)  | |-- start\_date: string (nullable = true) |

**Hint**:

Use from\_json() of pyspark

<https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.sql.functions.from_json.html#:~:text=Parses%20a%20column%20containing%20a,case%20of%20an%20unparseable%20string>.

Solution:

We will use pyspark.sql.function.from\_json()

**from\_json(col, schema, options={} )**

**col**: string column (*col, which is string but could be a valid json, python data*) in json format, mandatory

**schema**=a StructType or ArrayType of StructType to use when parsing the json column, mandatory

cam = {"emp\_id":1234, "emp\_acct": [6784, 8901], "start\_date":"2002-05-06"}

#Create the schema for for above value

schema=StructType([StructField("emp\_id", IntegerType()),\

                   StructField("emp\_acct", ArrayType(IntegerType() ) ),

                   StructField("start\_date", StringType() ) ])

#create new dataframe by using lit ans from\_json

df = df.withColumn("cam\_details", from\_json(lit(str(cam)), schema))

**Ouput**:

+---+--------------------------------+

|id |cam\_details |

+---+--------------------------------+

|1 |{1234, [6784, 8901], 2002-05-06}|

|2 |{1234, [6784, 8901], 2002-05-06}|

+---+--------------------------------+

root

|-- id: integer (nullable = true)

|-- cam\_details: struct (nullable = true)

| |-- emp\_id: integer (nullable = true)

| |-- emp\_acct: array (nullable = true)

| | |-- element: integer (containsNull = true)

| |-- start\_date: string (nullable = true)

**#Example # to\_json (StructType to string)**

Let say we have dataframe in which one column is MapType/StructType, now convert the column into string Type.

**to\_json( col\_name )**

cam = {"emp\_id":1234, "emp\_acct": [6784, 8901], "start\_date":"2002-05-06"}

#Create the schema for for above value

schema=StructType([StructField("emp\_id", IntegerType()),\

                   StructField("emp\_acct", ArrayType(IntegerType() ) ),

                   StructField("start\_date", StringType() ) ])

#create new dataframe by using lit, from\_json of StructType

df = df.withColumn("cam\_details", from\_json(lit(str(cam)), schema))

print(df.printSchema())

#Now convet back the cam\_details from StrucType to string Type

df = df.withColumn("cam\_details", to\_json(cam\_details))

print(df.printSchema())

**Observation:**

Observe the output difference between below(directly giving MapType for json string vs assigning schema for each key/col )

**Code1:**

#Create the schema for for above value

jsonString="""{"Zipcode":704,"ZipCodeType":"STANDARD","City":"PARC PARQUE","State":"PR"}"""

df=spark.createDataFrame([(1, jsonString)],["id","value"])

df.printSchema()

schema=StructType([StructField("Zipcode", IntegerType() ),\

                   StructField("ZipCodeType", StringType() ),

                   StructField("City", StringType() ),\

                  StructField("State", StringType() ) ])

#df = df.withColumn("cam\_details", from\_json(lit(str(cam)), schema))

df2=df.withColumn("value",from\_json(df.value,schema) )

df2.show(truncate=False)

df2.printSchema()

.

Code2

jsonString="""{"Zipcode":704,"ZipCodeType":"STANDARD","City":"PARC PARQUE","State":"PR"}"""

df=spark.createDataFrame([(1, jsonString)],["id","value"])

df.printSchema()

df2=df.withColumn("value",from\_json(df.value,MapType(StringType(),StringType())))

df2.printSchema()

df2.show(truncate=False)

Output

|  |  |
| --- | --- |
| **Code1**  |-- id: long (nullable = true)  |-- value: string (nullable = true)  **#after applying from\_json()**  root  |-- id: long (nullable = true)  **|-- value: struct (nullable = true)**  **| |-- Zipcode: integer (nullable = true)**  **| |-- ZipCodeType: string (nullable = true)**  **| |-- City: string (nullable = true)**  **| |-- State: string (nullable = true)** | **Code2**  root  |-- id: long (nullable = true)  **|-- value: string (nullable = true)**  **#after applying from\_json()**  root  |-- id: long (nullable = true)  **|-- value: map (nullable = true)**  **| |-- key: string**  **| |-- value: string (valueContainsNull = true)** |

**Note:**

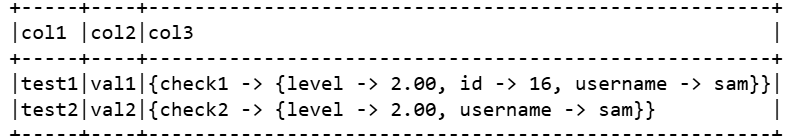
* When we give proper schema ( key type and value type ) then it’s not showing key name in result.
* When we don’t give proper schema ( key type and value type ) and just given MapType() then it’s showing key-value in result.

**#Question# -- explode on maptype col**

We have input dataframe, where one of column ( col3) contains MapType(Maptype() ) data.

Now for each key-value pair create one row.

**Input dataframe:**



schema=StructType([StructField('col1',StringType(),True),StructField('col2',StringType(),True),\

StructField('col3',MapType(StringType(),MapType(StringType(),StringType()),True))])

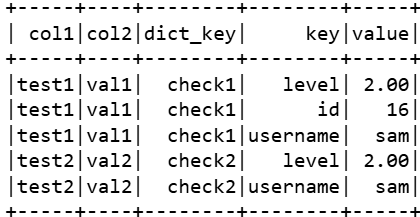
data=[('test1','val1', {'check1':{ "id": "16","username":"sam","level":"2.00"}}),

('test2','val2',{'check2':{"username":"sam","level":"2.00"}})]

df=spark.createDataFrame(data=data,schema=schema)

df.show(truncate=False)

**Output**



**Hint:**

Use explode function (explode can be used on Map and Array )

**working of explode() on maptype from below example**

|  |  |
| --- | --- |
| **Input**    **#cahnging the col name#**  **df.select('col1','col2',explode('col3').\**  **alias('my\_name','my\_value') )**  **Note:**  Using alias('X','Y') function names are changed | **#default new col name#**  **df.select('col1','col2',explode('col3')).show()**    **Note:**  By default new column name are 'key' and 'value' |

**Solution:**

schema=StructType([StructField('col1',StringType(),True),\

                   StructField('col2',StringType(),True),\

                 StructField('col3',MapType(StringType(),MapType(StringType(),StringType()),True))])

data=[('test1','val1', {'check1':{ "id": "16","username":"sam","level":"2.00"}}),

     ('test2','val2',{'check2':{"username":"sam","level":"2.00"}})]

df=spark.createDataFrame(data=data,schema=schema)

df.show(truncate=False)

df.printSchema()

#alias('dict\_key','dict\_value') does below work

#cahnges default column name -- key to dict\_key

#cahnges default column name -- value to dict\_value

**df=df.select('col1','col2',explode(df.col3).alias('dict\_key','dict\_value'))**

df.show()

df.select('col1','col2','dict\_key',explode(df.dict\_value)).show()

**###########################**

**# explode() vs explode\_outer() #**

**###########################**

|  |  |
| --- | --- |
| **explode()**   * if create new row for each value of array/list/map type column data * If map/array/list type column contains null then new row is not created | **explode\_outer()**   * if create new row for each value of array/list/map type column data * If map/array/list type column contains null then new row is created/retained |

**Let take below xample:**

Let say we have below dataframe as –

|  |  |
| --- | --- |
| data = [("John", [1, 2, 3]), ("Alice", [4, 5]), ("Bob", None)]  columns = ["name", "numbers"]  df = spark.createDataFrame(data, columns)  df.show() |  |

|  |  |
| --- | --- |
| **explode()**  **df.select("name", explode(df.numbers))** | **explode\_outer()**  df.select("name", explode\_outer(df.numbers)) |

#Question# --- aggregation on window

<https://stackoverflow.com/questions/79084670/create-column-of-current-balance-in-pyspark/79084677#79084677>

We have given dataframe like below and we have to calculate the ( the balance for each account number ) as below

|  |  |
| --- | --- |
| Input    data=[('2023-01-01',100,'Credit',1000),('2023-01-02',100,'Credit',1500),('2023-01-03',100,'Debit',1000),\  ('2023-01-02',200,'Credit',3500),('2023-01-03',200,'Debit',2000),('2023-01-04',200,'Credit',3500),\  ('2023-01-13',300,'Credit',4000),('2023-01-14',300,'Debit',4500),('2023-01-15',300,'Credit',5000)]  schema=['TransactionDate','AccountNumber','TransactionType','Amount']  df=spark.createDataFrame(data=data,schema=schema)  df.show(truncate=False) | Ouput/expected |

Hint:

* Use sum() with window on account number order by transaction date
* More about aggregation with window is on below link-

<https://medium.com/@uzzaman.ahmed/pyspark-aggregate-window-functions-a-comprehensive-guide-f417b6962424#:~:text=An%20aggregate%20window%20function%20in,in%20that%20group%20of%20rows>.

Solution:

df=df.withColumn('cal\_amount',when ( col('TransactionType')=='Credit',col('Amount')\*1 ).otherwise(-1\*col('Amount')))

df=df.withColumn('TransactionDate',to\_date( col('TransactionDate'),'yyyy-MM-dd' ))

df.show()

window = Window.partitionBy('AccountNumber').orderBy('TransactionDate')

df = df.withColumn('CurrentBalance', max("cal\_amount").over(window))

df.show()

**Question – Cumulative sum in a dataframe grouped by year-month-date**

This is actual question *from stackoverflow for - grouped by year-month but here year-month-date*

<https://stackoverflow.com/questions/79448041/cumulative-sum-in-a-dataframe-grouped-by-year-month>

We have one dataframe which contains the release date, calculate the release countn for each date in column rel\_count and commulative release count in column cum\_rel\_count.

|  |  |
| --- | --- |
| Input Dataframe  +----------+  | rel\_dt|  +----------+  |1997-06-30|  |1997-11-14|  |1998-11-08|  |1999-04-01|  |1999-09-08|  |1999-11-01|  |2000-11-01|  |2000-11-01|  |2001-03-15|  |2001-06-01|  |2001-06-01|  +----------+  data=[('1997-06-30',),('1997-11-14',),('1998-11-08',),('1999-04-01',),('1999-09-08',),  ('1999-11-01',),('2000-11-01',),('2000-11-01',),('2001-03-15',),('2001-06-01',),('2001-06-01',)]  schema=StructType([StructField('rel\_dt',StringType(),True)])  df=spark.createDataFrame(data=data,schema=schema) | Output  +----------+---------+-------+  | rel\_dt|rel\_count|cum\_sum|  +----------+---------+-------+  |1997-06-30| 1| 1|  |1997-11-14| 1| 2|  |1998-11-08| 1| 3|  |1999-04-01| 1| 4|  |1999-09-08| 1| 5|  |1999-11-01| 1| 6|  |2000-11-01| 2| 8|  |2001-03-15| 1| 9|  |2001-06-01| 2| 11|  +----------+---------+-------+ |

**Hint:**

* First calculate the release count by using groupby and count
* Then create the window function to calculate the cum sum

Solution:

df=spark.createDataFrame(data=data,schema=schema)

df.show()

df=df.groupBy(df.rel\_dt).count()

df=df.withColumnRenamed('count','rel\_count')

spec=Window.orderBy(df.rel\_dt).rangeBetween(Window.unboundedPreceding, 0)

df=df.withColumn('cum\_sum',sum('rel\_count').over(spec))

df.show()

**#Question# -- Create MapType from dataframe column**

<https://stackoverflow.com/questions/78637317/is-there-a-way-to-store-a-dictionary-as-a-column-value-in-pyspark>

Let say we have one dataframe having columns – col1, col2, col3, … coln. For columns col1, col2 create a new column name – map\_type\_col which will hold data of col1, col2 as key-value pair

|  |  |
| --- | --- |
| Input | Output |

#Input df#

schema = StructType([StructField("col1", StringType(), True),\

                     StructField("col2", IntegerType(), True),\

                     StructField("col3", IntegerType(), True)])

# Define the data

data = [ ("AAAAAAAA", 77836, 3243),("BBBBBBBB", 896245, 3455),\

        ("CACACA", None, -983),("DHVDSR", None, 3455),\

        (None, 77652, None)]

# Create the DataFrame

df = spark.createDataFrame(data, schema)

**Solution:**

df.withColumn('map\_type\_col',create\_map(lit('col1'),df.col1,lit('col2'),df.col2))

Note:

For each input argument of create\_map()-

First argument : --- key

Second argument : -- value of first key (First argument)

Second argument : --- key

Second argument : -- value of first key (Second argument)

And so on

**#Question# Example on json\_tuple**

For below dataframe, create new column for each age and city key from the json\_str for each row.

|  |  |
| --- | --- |
| root  |-- id: string (nullable = true)  |-- json\_str: string (nullable = true) | Output |

#Input df#

data = [("1", '{"name":"John", "age":30, "city":"New York"}'),

("2", '{"name":"Alice", "age":25, "city":"Los Angeles"}'),

("3", '{"name":"Bob", "age":35, "city":"Chicago"}')]

columns = ["id", "json\_str"]

df = spark.createDataFrame(data, columns)

df.show(truncate=False)

**Solution:**

extracted\_df = df.select("id", json\_tuple("json\_str","age", "city").alias( "age", "city"))

# Show the result

extracted\_df.show()

################################################################################

# Interview/Learned questions #

################################################################################

**Question:**

We have two dataframe then **concatenate both dataframe on columns wise**.

Keep in mind that there **is no column which can be considered as PKEY/UNIQUE column**

**Hint:**

Convert into RDD and then apply the zip function then map function to change data of each zip object/row into list and change back into dataframe

**Solution:**

I am assuming that I have two dataframe in which each dataframe have three columns.

#this finction is used to create a row data in list form

#the data will be derived from input variable x

def myFunc(x):

    dt1 = x[0]

    dt2 = x[1]

    col1=dt1[0]

    col2=dt1[1]

    col3=dt1[2]

    col11=dt2[0]

    col22=dt2[1]

    col33=dt2[2]

    return [col1,col2,col3,col11,col22,col33]

#create schema from both dataframes schema

df3\_schema = StructType(df1.schema.fields + df2.schema.fields)

#Conver both datafraem into RDD and apply zip function on them

rdd\_new=df1.rdd.zip(df2.rdd).map(lambda x: myFunc(x))

#now conver the rdd into dataframe

df3=spark.createDataFrame(df3, schema=df3\_schema)

print(df3.show())

**Note:**

We can use above approach to compare the psyaprk dataframe element wise

**Note:**

If we are converting any string into timestamp in pyspark on pandas then string should be such that if it is converted into date value > 22-Sept-1677.

**What is spark?**

Apache Spark is an open-source, **distributed processing system** used for big data workloads.

**Spark is a fast** and general engine for large-scale data processing.

The fast part means that it’s faster than previous approaches to work with Big Data like classical MapReduce.

**The secret for being faster is that Spark runs on memory (RAM)**, and that makes the processing much faster than on disk drives.

**What is Distributed Processing System.**

In Distributed system huge amount of data is split across multiple systems or servers, and processed in a distributed manner. This is called distributed computing and Apache Spark is one of the open-source frameworks for distributed computing

**What is in-memory processing in Spark?**

In in-memory computation, the data is kept in random access memory(RAM) instead of some disk drives and is processed in parallel.

In **memory processing makes the processing faster**.

**What is RDD?**

RDDs are the main logical data units in Spark. They are a distributed collection of objects, which are stored in memory or on disks of different machines of a cluster.

A single RDD can be divided into multiple logical partitions so that these partitions can be stored and processed on different machines of a cluster.

In Spark all data DataFrame, RDD or anything internally used RDD.

**What is Spark Session?**

Sparksession was introduced in version Spark 2.0.

By default, it is created by name 'spark'

It is an entry point to underlying Spark functionality in order to programmatically create Spark RDD, DataFrame, and DataSet.

The entry point to programming Spark with the Dataset and DataFrame API.

A Sparksession can be used create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read files.

Prior to Spark 2.0, SparkContext used to be entry point.

**SparkConext**: Entry point to work with RD, Accumulator and broadcast variables, by default - sc

**SQLContext**: Used for initializing the functionalities for Spark SQL. Used to create RDD

**HiveContext**: Super set of SQLContext.

**What are RDDs in PySpark?**

RDDs expand to Resilient Distributed Datasets.

These are the elements that are used for running and operating on multiple nodes to perform parallel processing on a cluster.

Since RDDs are suited for parallel processing, they are immutable elements.

Means that once we create RDD, we cannot modify it. Any changes done on RDD creates new RDD.

RDDs are also fault-tolerant which means that whenever failure happens, they can be recovered automatically. Multiple operations can be performed on RDDs to perform a certain task.



**RDD operation can be of 2 types:**

1. Transformation
2. Action

**Transformation Operation:**

These operations when applied on RDDs result in the creation of a new RDD

**Action Operation:**

These operations instruct Spark to perform some computations on the RDD and return the result but does not create new RDD to the driver.

**Difference between Pandas and PySpark DataFrame**



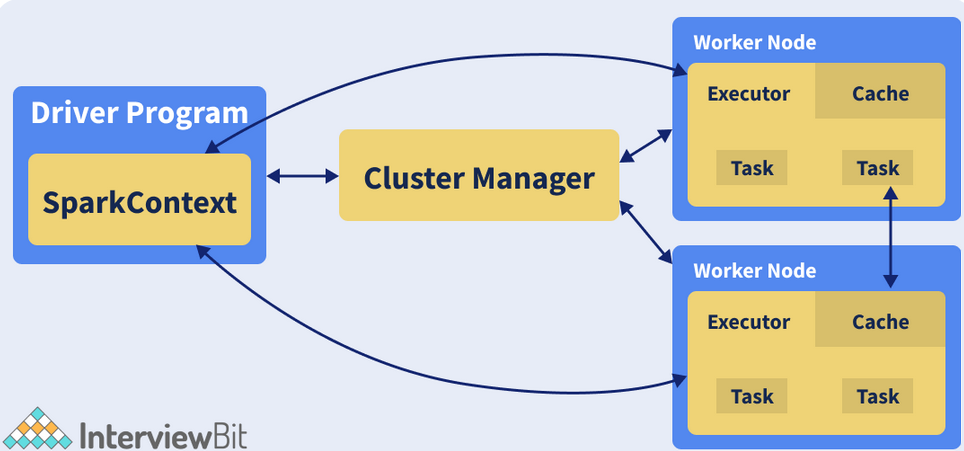
**Advantage of Pyspark**

**What is PySpark UDF?**

UDF stands for User Defined Function. It is used when there is requirement of operation that cannot be done by pyspark function or we have to do that operation frequently in such case we can defined own function and register it for usage.

**What is PySpark Architecture?**

PySpark similar to Apache Spark works in master-slave architecture pattern. Here, the master node is called the Driver and the slave nodes are called the workers. When a Spark application is run, the Spark Driver creates SparkContext which acts as an entry point to the spark application. All the operations are executed on the worker nodes. The resources required for executing the operations on the worker nodes are managed by the Cluster Managers. The following diagram illustrates the architecture described:



**Different way of creating RDD**

We can create RDD from below different way-

1. From Externa Data ( HDFS, local etc )
2. Local Data
3. Python List/ Parallelized collection
4. Other RDD
5. Existing DataFrame

**What is difference between StructType and MapType in pyspark**

**StructType**: When a column of pyspark have fixed sub-column(name) and fixed data type of sub-column value for each row.

**MapType**: When column of pyspark don't have any sub-column(name) then use MapType

|  |  |
| --- | --- |
| MapType    Utilities column have sub-column but are dynamic. | StructType    *Name column have 3 sub-column (FirstName, MiddleName, LastName) and will have some value(including null) for each row of Name column*. |

**Different way of creating DataFrame**

1. From File
2. From Database table
3. From RDD
4. From HDFS files etc

**#Example# --- concatenating/placing two dataframe**

Perform concatenating/placing two dataframe side by side

**Hint**:

**Assign index column** in both dataframe

Now perform **inner join** on index column

**What are the key differences between RDDs, DataFrames, and Datasets in PySpark?**

Spark Resilient Distributed Datasets (RDD), DataFrame, and Datasets are key abstractions in Spark that enable us to work with structured data in a distributed computing environment. Even though they are all ways of representing data, they have key differences:

* RDDs are low-level APIs that lack a schema and offer control over the data. They are immutable collections of objects.
* DataFrames are high-level APIs built on top of RDDs optimized for performance but are not safe-type. They organize structured and semi-structured data into named columns.
* Datasets combine the benefits of RDDs and DataFrames. They are high-level APIs that provide safe-type abstraction. They support Python and Scala and provide compile-time type checking while being faster than DataFrames.



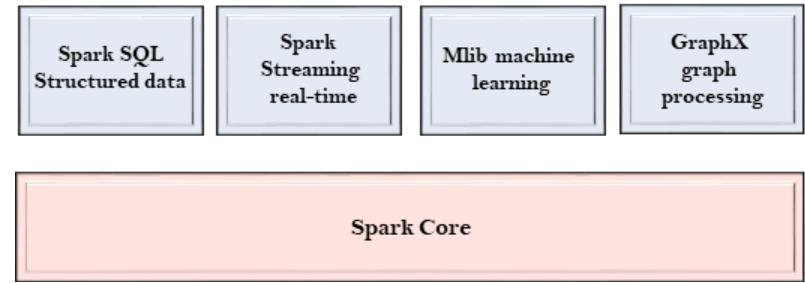
**What is Lazy evaluation.**

PySpark implements a strategy called lazy evaluation, where the transformations applied on distributed datasets (RDDs, DataFrames, or Datasets) are not executed immediately. On the contrary, Spark builds a sequence of operations or transformations to be performed on the data called a directed acyclic graph (DAG).

This lazy evaluation improves performance and optimizes execution because the computation is deferred until an action is triggered and strictly necessary

**Explain is spark eco system**

Following are components in Apache Spark Ecosystem which empower to Apache Spark.



**Spark Core**

* The Spark Core is the heart of Spark and performs the core functionality
* It holds the components for task scheduling, fault recovery, interacting with storage systems and memory management.
* Spark Core is the foundation of parallel and distributed processing of huge dataset.
* The key features of Apache Spark Core are:

1. It is in charge of essential I/O functionalities.
2. Significant in programming and observing the role of the Spark cluster.
3. Task dispatching.
4. Fault recovery.
5. It overcomes the snag of MapReduce by using in-memory computation.

**Spark SQL**

* The Spark SQL is built on the top of Spark Core. It provides support for structured data.
* Using Spark SQL, Spark gets more information about the structure of data and the computation and perform extra optimization.
* It uses same execution engine while computing an output. It does not depend on API/ language to express the computation.

**Spark Streaming**

* Spark Streaming is a Spark component that supports scalable and fault-tolerant processing of streaming data.
* It uses Spark Core's fast scheduling capability to perform streaming analytics.
* It accepts data in mini-batches and performs RDD transformations on that data.
* Its design ensures that the applications written for streaming data can be reused to analyze batches of historical data with little modification.
* The log files generated by web servers can be considered as a real-time example of a data stream.

**MLlib**

* The MLlib is a Machine Learning library that contains various machine learning algorithms.
* These include correlations and hypothesis testing, classification and regression, clustering, and principal component analysis.
* It is nine times faster than the disk-based implementation used by Apache Mahout.

**GraphX**

* The GraphX is a library that is used to manipulate graphs and perform graph-parallel computations.
* It facilitates to create a directed graph with arbitrary properties attached to each vertex and edge.
* To manipulate graph, it supports various fundamental operators like subgraph, join Vertices, and aggregate Messages.

**What is broadcast variable**

As name suggest, they are read-only shared variables that are cached and distributed to the cluster nodes to avoid shuffle operations.

In PySpark Broadcast is created using the broadcast(v) method of the SparkContext class method.

Broadcast variables are not sent to executors with sc.broadcast(variable) call instead, they will be sent to executors when they are first used.

**Example of broadcast variable**

import pyspark

from pyspark.sql import SparkSession

#

spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

#create the broadcast variable and broadcast it

states = {"NY":"New York", "CA":"California", "FL":"Florida"}

broadcastStates = spark.sparkContext.broadcast(states)

#now create the RDD

data = [("James","Smith","USA","CA"),("Michael","Rose","USA","NY"),("Robert","Williams","USA","CA"), ("Maria","Jones","USA","FL")]

rdd = spark.sparkContext.parallelize(data)

print(rdd.take(10))

#create function to return the name nased on country code

def state\_convert(code):

    return broadcastStates.value[code]

#

result = rdd.map(lambda x: (x[0],x[1],x[2],state\_convert(x[3]))).collect()

print(result)

**Explain the Driver/Supervisor and executor.**

**Difference between map and flatmap in pyspark**

**##################**

**# map() vs flatmap() #**

**##################**

*Check filter and map in python function also****.***

<https://medium.com/@sujathamudadla1213/what-is-the-difference-between-map-and-flatmap-in-spark-6e6fa9360094>

|  |  |
| --- | --- |
| Flatmap | map |
| In flatmap applies on each element of input RDD/Dataframe and returns new RDD/Dataframe where each input element can be mapped to zero or more output elements | In flatmap applies on each element of input RDD/Dataframe and returns RDD/Dataframe where each element corresponds to the result of applying the function to the corresponding element of the original RDD or DataFrame |
| flatMap can return output a sequence or iterable of values for each input element, and these values will be concatenated into the output RDD or DataFrame i.e output will be one iterable data | The output of the map transformation has a one-to-one mapping between input and output elements. In other words, if you start with an RDD or DataFrame of n elements, the map transformation will produce a new RDD or DataFrame with the **same number of elements** ( each input will have one output data which can be iterable also) |

**#Question# --- LeetCode**

<https://stackoverflow.com/questions/79495236/apply-pivot-only-on-some-columns-in-pyspark/79496813#79496813>

Grouping, aggregate and select first data from column.

|  |  |
| --- | --- |
| Input dataframe    df = spark.createDataFrame(  [[1, 'AB', 12, '2022-01-01'], [1, 'AA', 22, '2022-01-10'], [1, 'AC', 11, '2022-01-11'], [2, 'AB', 22, '2022-02-01'],\  [2, 'AA', 28, '2022-02-10'], [2, 'AC', 25, '2022-02-22']], 'code: int, doc\_type: string, amount: int, load\_date: string'  )  df=df.withColumn('load\_date',to\_date(df.load\_date,'yyyy-MM-dd'))  df.show() | Output dataframe |

**Solution:**

df=df.withColumn('load\_date',to\_date(df.load\_date,'yyyy-MM-dd'))

df.show()

df1=df.groupBy('code').pivot(pivot\_col='doc\_type',values=['AB', 'AA', 'AC']).agg(sum('amount'))

df1.show()

df2=df.groupBy('code').agg(first('load\_date').alias('load\_date1'))

df2.show()

df\_result=df1.join(df2,[df1.code==df2.code],how='outer').select(df1.code,df1.AB,df1.AA,df1.AC,df2.load\_date1)

df\_result.show()

**#Question#**

Keep first ocuurance and delete other occurance of col from df for duplicate col name.

<https://www.geeksforgeeks.org/drop-a-column-with-same-name-using-column-index-in-pyspark/>

|  |  |
| --- | --- |
| Input | Output |

df = spark.createDataFrame(

[('Monday',25,27,29,30),('Tuesday',40,38,36,34),

('Wednesday',18,20,22,17),('Thursday',25,27,29,19)],

['day','temperature','temperature','temperature', 'temperature'])

# Store all the column names in the list

df\_cols = df.columns

# Get index of the duplicate columns

duplicate\_col\_index = [idx for idx, val in enumerate(df\_cols) if val in df\_cols[:idx]]

# Create a new list by renaming duplicate columns by adding prefix 'day\_'

for i in duplicate\_col\_index:

    df\_cols[i] = 'day\_'+ df\_cols[i]

# Rename the duplicate columns in data frame

df = df.toDF(\*df\_cols)

#Now drop the column which are duplicate and renamed