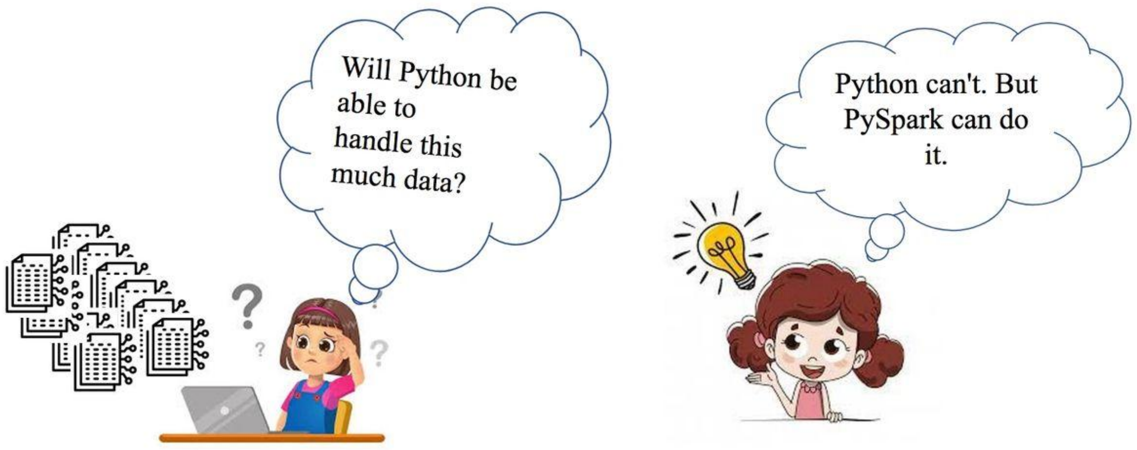
Let's Start...!!
The goal of this post is to discuss PySpark, which is one of the key members of Big data analytics, and discuss a few basic PySpark analytical functionalities. Extraction of benefits from large data is a bit tricky and it needs efficient tools to play with this data ocean. PySpark is one of the important tools available and is high on demand.


Will Python be able to handle this much data?
Python can't. But PySpark can do it.

Edit Image
PySpark can handle big data analytics efficiently compared to Python
Python
Python is a simple, easy to learn, powerful, high level and object-oriented programming language. Guido Van Rossum is known as the founder of Python programming. Now Python is one of the popular languages in use across different areas like web development, Artificial Intelligence, and Machine Learning & many Enterprise-level/Business Applications, etc.
Spark
Spark is an open-source distributed engine designed for real-time large-scale data processing. Spark is becoming an important component of many architectures for large-scale data analysis.

Edit Image
Where do Pandas surrender?
Pandas is a library written for the Python programming language for data manipulation and analysis. But it will fail to perform in the following situations.

1. Large Data size - When the data size is too large to handle by python.
2. For real-time streaming data - When the data is from Streaming or live sessions.
3. Distributed data - When data is stored across multiple computers, we need to integrate everything for analytics. Pandas have limitations to perform in this situation.

What is the Solution??????

We need to walk away from Pandas in the above situations and PySpark is an alternative. As we discussed above Spark can handle big data and Python has its own features in analytics. And PySpark is the platform that integrates and gives all benefits of Python & Spark.

Edit Image

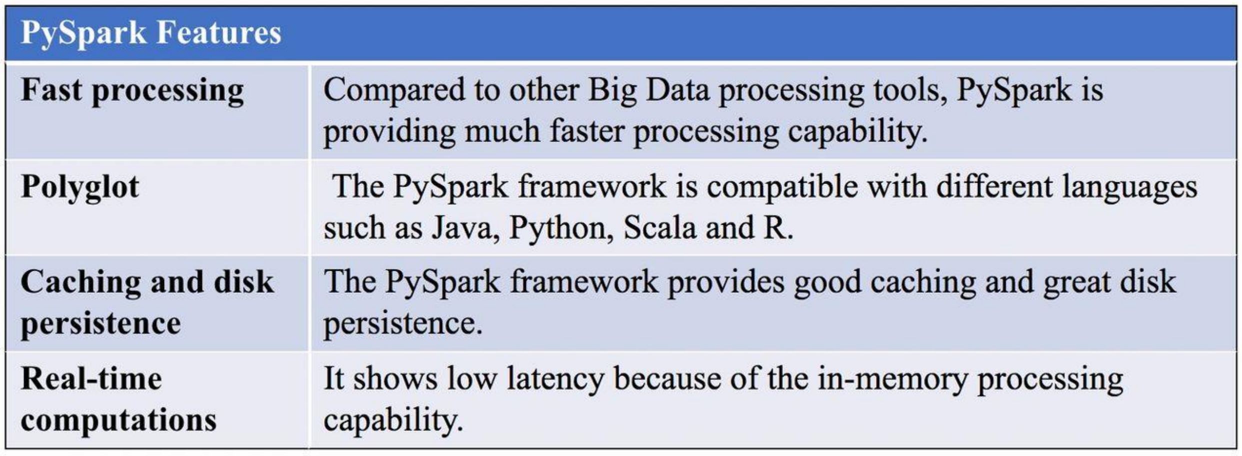
Python + Spark = PySpark

PySpark

As the name suggests, PySpark is a Python interface for Apache Spark. This Python API supports Python integration with Spark for analyzing large data in a distributed environment and it is becoming one of the major demanding tools because of its high efficiency in dealing with large data. It supports a majority of Spark's features such as MLlib, DataFrame & SQL, etc.

PySpark installation: http://spark.apache.org/docs/latest/api/python/getting_started/install.html

Key features of PySpark



PySpark Features	
Fast processing	Compared to other Big Data processing tools, PySpark is providing much faster processing capability.
Polyglot	The PySpark framework is compatible with different languages such as Java, Python, Scala and R.
Caching and disk persistence	The PySpark framework provides good caching and great disk persistence.
Real-time computations	It shows low latency because of the in-memory processing capability.

Edit Image

PySpark Functions

Let's discuss core PySpark functionalities in detail.

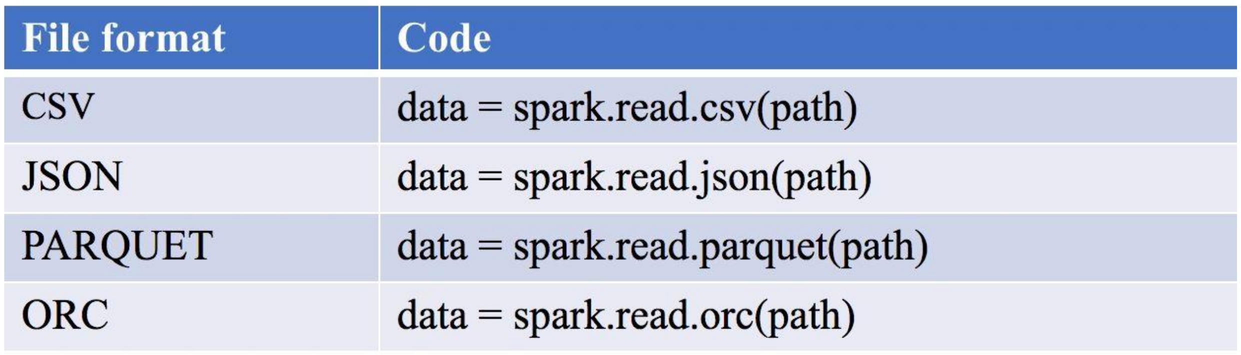
Reading data

Considering Iris's data set in the following sessions to explain the examples.

data source: https://archive.ics.uci.edu/ml/datasets/iris or https://gist.github.com/netj/8836201

The PySpark can read different file formats such as CSV (Comma Separated Values), Parquet, JSON (JavaScript Object Notation), ORC, AVRO, etc.

Examples: Read the different files from the given path



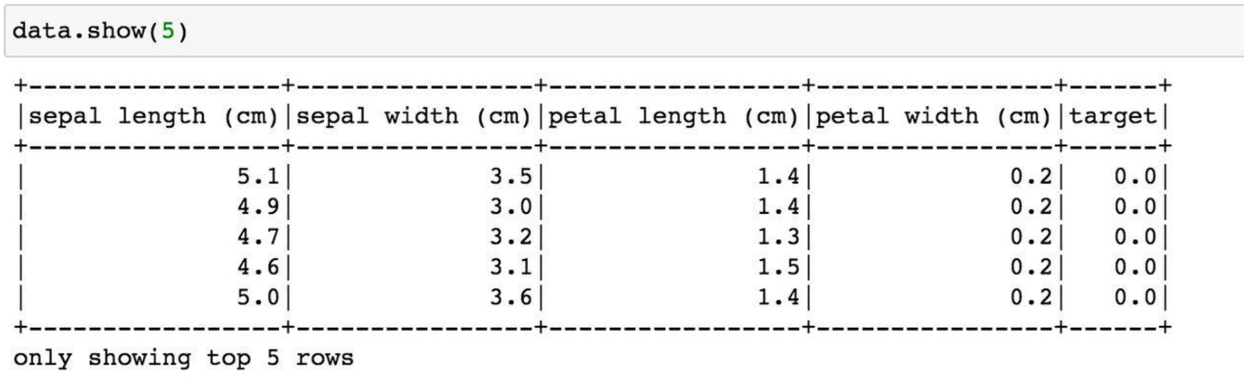
File format	Code
CSV	data = spark.read.csv(path)
JSON	data = spark.read.json(path)
PARQUET	data = spark.read.parquet(path)
ORC	data = spark.read.orc(path)

Edit Image

Understand the data structure

show()

Function show(n) prints the first n rows in the console in a table format. The default value of n equals 20.



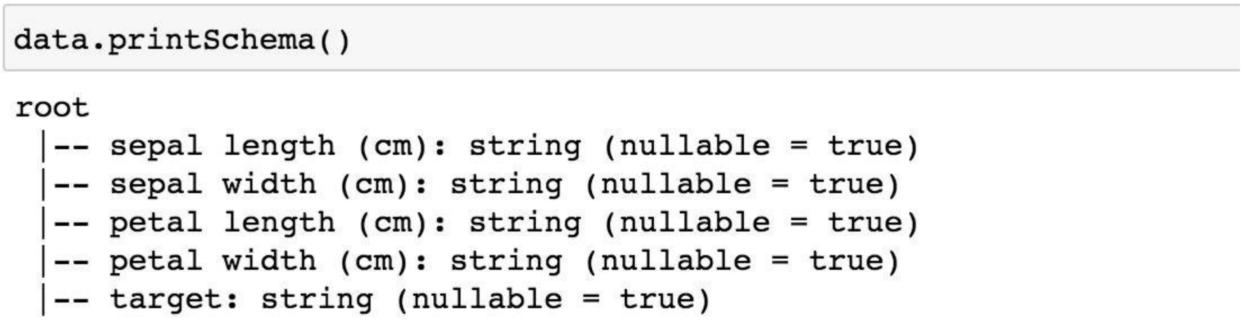
```
data.show(5)
```

```
+-----------------+----------------+-----------------+----------------+------+
|sepal length (cm)|sepal width (cm)|petal length (cm)|petal width (cm)|target|
+-----------------+----------------+-----------------+----------------+------+
|              5.1|             3.5|              1.4|             0.2|   0.0|
|              4.9|             3.0|              1.4|             0.2|   0.0|
|              4.7|             3.2|              1.3|             0.2|   0.0|
|              4.6|             3.1|              1.5|             0.2|   0.0|
|              5.0|             3.6|              1.4|             0.2|   0.0|
+-----------------+----------------+-----------------+----------------+------+
only showing top 5 rows
```

Edit Image

printSchema()

Function printSchema() prints the schema of the data frame in a tree format.



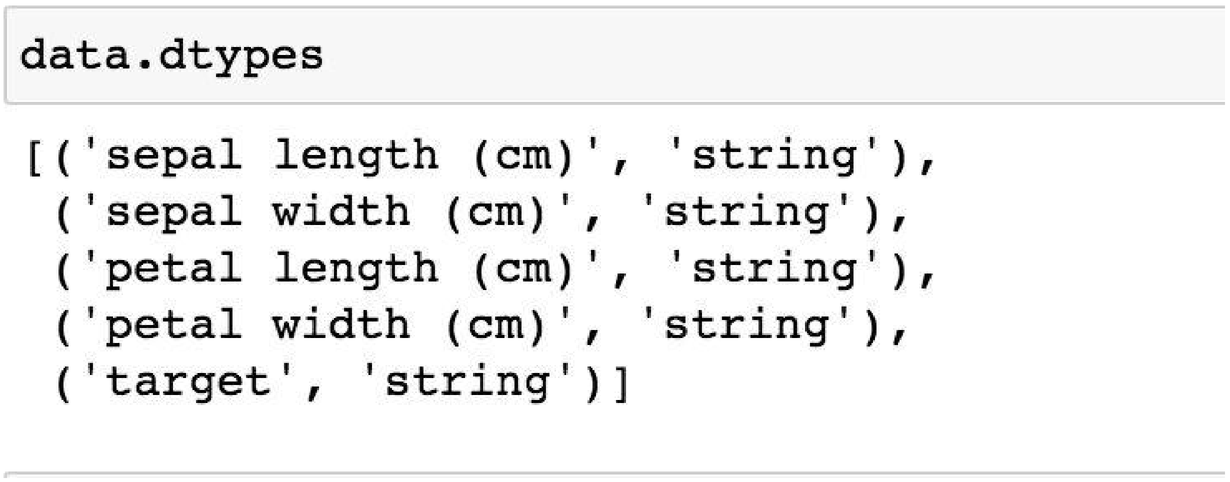
```
data.printSchema()
```

```
root
 |-- sepal length (cm): string (nullable = true)
 |-- sepal width (cm): string (nullable = true)
 |-- petal length (cm): string (nullable = true)
 |-- petal width (cm): string (nullable = true)
 |-- target: string (nullable = true)
```

Edit Image

dtypes

dtypes returns all column names and their data types as a list.



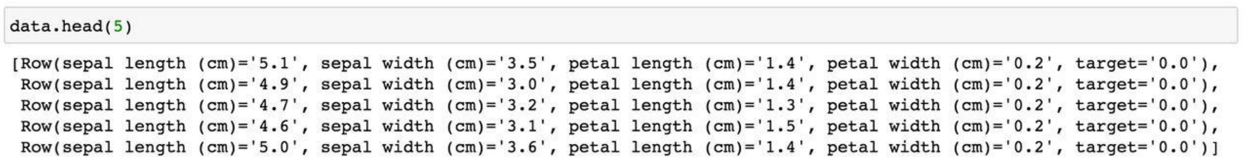
```
data.dtypes
```

```
[('sepal length (cm)', 'string'),
 ('sepal width (cm)', 'string'),
 ('petal length (cm)', 'string'),
 ('petal width (cm)', 'string'),
 ('target', 'string')]
```

Edit Image

head()

Function head(n) returns the first `n` rows.



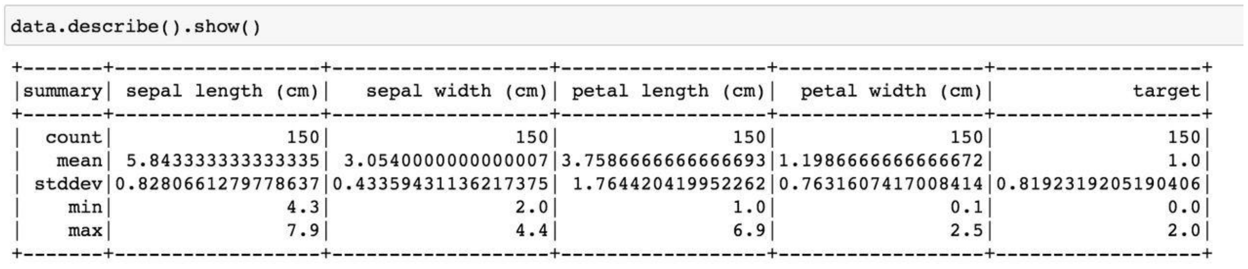
```
data.head(5)
```

```
[Row(sepal length (cm)='5.1', sepal width (cm)='3.5', petal length (cm)='1.4', petal width (cm)='0.2', target='0.0'),
 Row(sepal length (cm)='4.9', sepal width (cm)='3.0', petal length (cm)='1.4', petal width (cm)='0.2', target='0.0'),
 Row(sepal length (cm)='4.7', sepal width (cm)='3.2', petal length (cm)='1.3', petal width (cm)='0.2', target='0.0'),
 Row(sepal length (cm)='4.6', sepal width (cm)='3.1', petal length (cm)='1.5', petal width (cm)='0.2', target='0.0'),
 Row(sepal length (cm)='5.0', sepal width (cm)='3.6', petal length (cm)='1.4', petal width (cm)='0.2', target='0.0')]
```

Edit Image

describe()

Function describe() computes basic statistics (count, mean, standard deviation, minimum value, maximum value) across the possible columns.

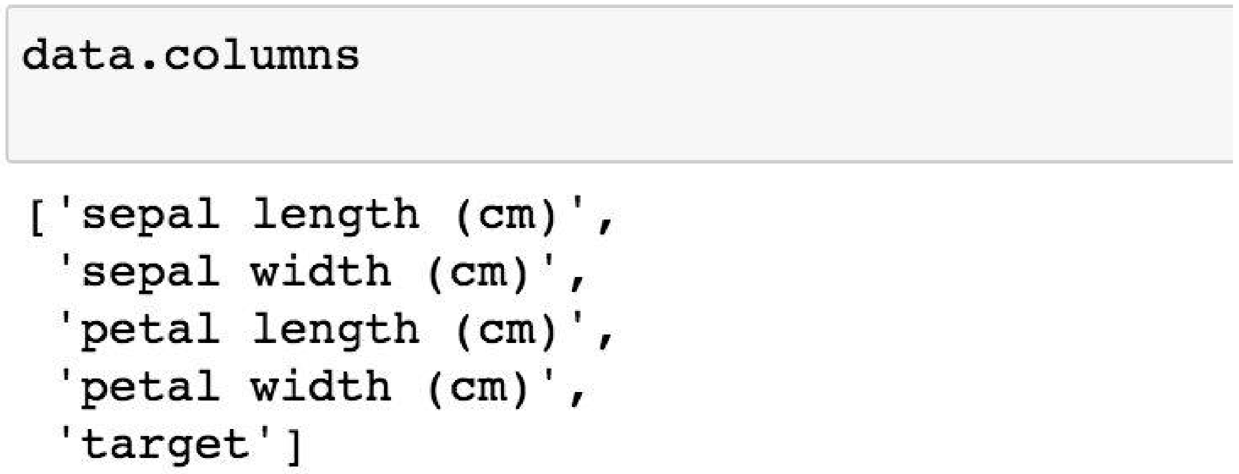


```
data.describe().show()
```

```
+-------+------------------+-------------------+------------------+------------------+------------------+
|summary| sepal length (cm)|   sepal width (cm)| petal length (cm)|  petal width (cm)|            target|
+-------+------------------+-------------------+------------------+------------------+------------------+
|  count|               150|                150|               150|               150|               150|
|   mean| 5.843333333333335| 3.0540000000000007|3.7586666666666693|1.1986666666666672|               1.0|
| stddev|0.8280661279778637|0.43359431136217375| 1.764420419952262|0.7631607417008414|0.8192319205190406|
|    min|               4.3|                2.0|               1.0|               0.1|               0.0|
|    max|               7.9|                4.4|               6.9|               2.5|               2.0|
+-------+------------------+-------------------+------------------+------------------+------------------+
```

Edit Image

columns

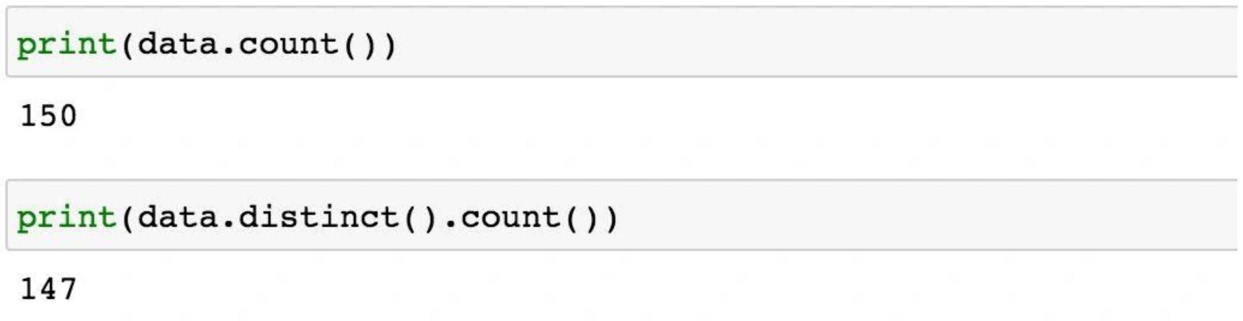


```
data.columns
```

```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)',
 'target']
```

Edit Image
columns return all column names as a list.

count()
count() returns the number of rows in a data frame and distinct().count() returns the number of distinct rows in a data frame.



```
print(data.count())
```

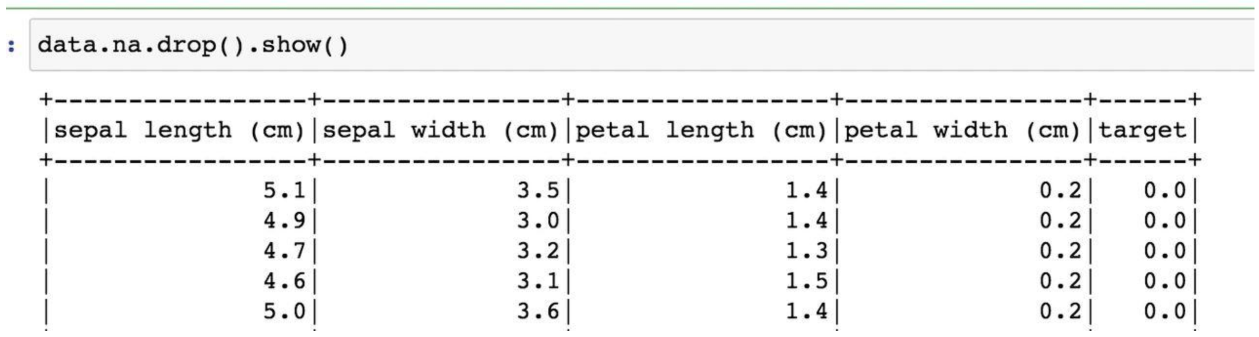
```
150
```

```
print(data.distinct().count())
```

```
147
```

Edit Image

Dealing with Missing values
In real-world data, missing values in some attributes are common, which are encountered for multiple reasons such as manual errors in recording, loss of information, etc. These missing values are commonly encoded as Blank spaces and NaN. Since these records are not adding any values or will cause misleading in the analysis procedure, it is necessary to handle these records at the beginning of the analysis itself. Following are the popular techniques to deal with missing values in the data.
Remove records with nan / missing values

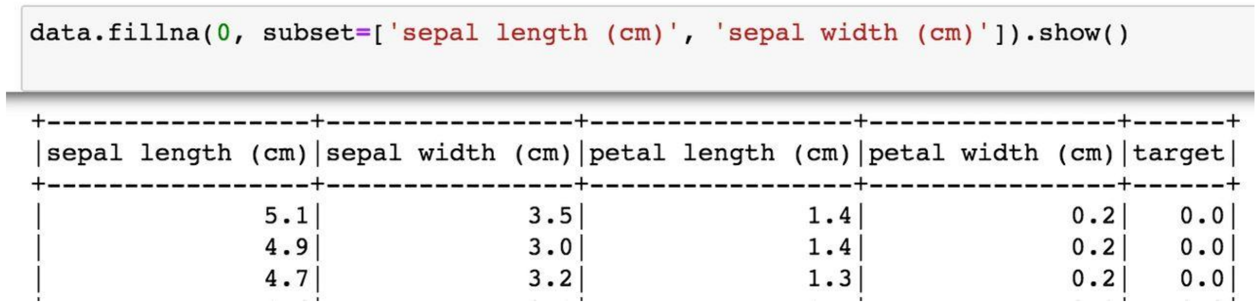


```
data.na.drop().show()
```

```
+-----------------+----------------+-----------------+----------------+------+
|sepal length (cm)|sepal width (cm)|petal length (cm)|petal width (cm)|target|
+-----------------+----------------+-----------------+----------------+------+
|              5.1|             3.5|              1.4|             0.2|   0.0|
|              4.9|             3.0|              1.4|             0.2|   0.0|
|              4.7|             3.2|              1.3|             0.2|   0.0|
|              4.6|             3.1|              1.5|             0.2|   0.0|
|              5.0|             3.6|              1.4|             0.2|   0.0|
```

Edit Image

Fill nan / missing values with specific values

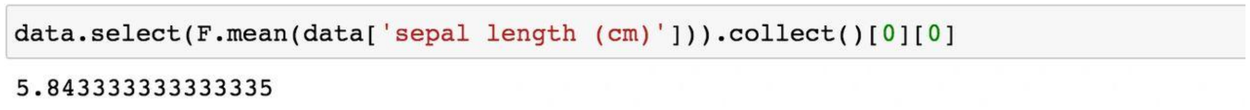


```
data.fillna(0, subset=['sepal length (cm)', 'sepal width (cm)']).show()
```

```
+-----------------+----------------+-----------------+----------------+------+
|sepal length (cm)|sepal width (cm)|petal length (cm)|petal width (cm)|target|
+-----------------+----------------+-----------------+----------------+------+
|              5.1|             3.5|              1.4|             0.2|   0.0|
|              4.9|             3.0|              1.4|             0.2|   0.0|
|              4.7|             3.2|              1.3|             0.2|   0.0|
```

Edit Image

Replace nan / missing values with mean



```
data.select(F.mean(data['sepal length (cm)'])).collect()[0][0]
```

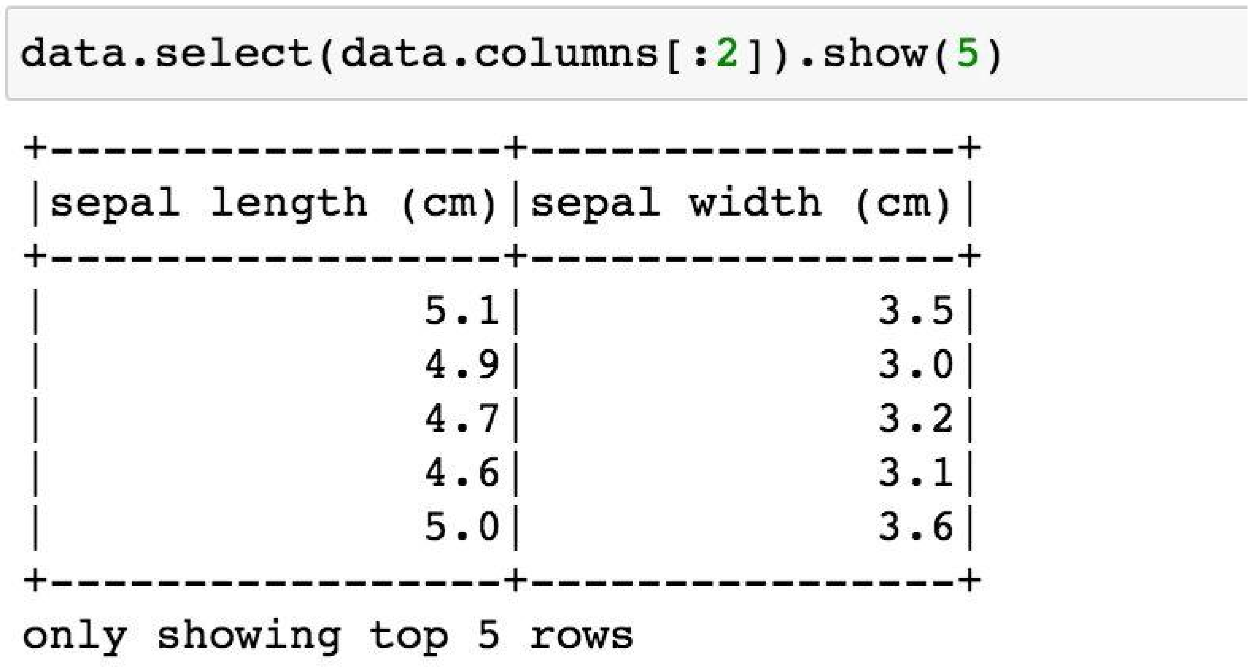
```
5.843333333333335
```

Edit Image

Data Inspection

There are different ways to inspect columns and rows in a data frame and the following sections are discussing a few of them.

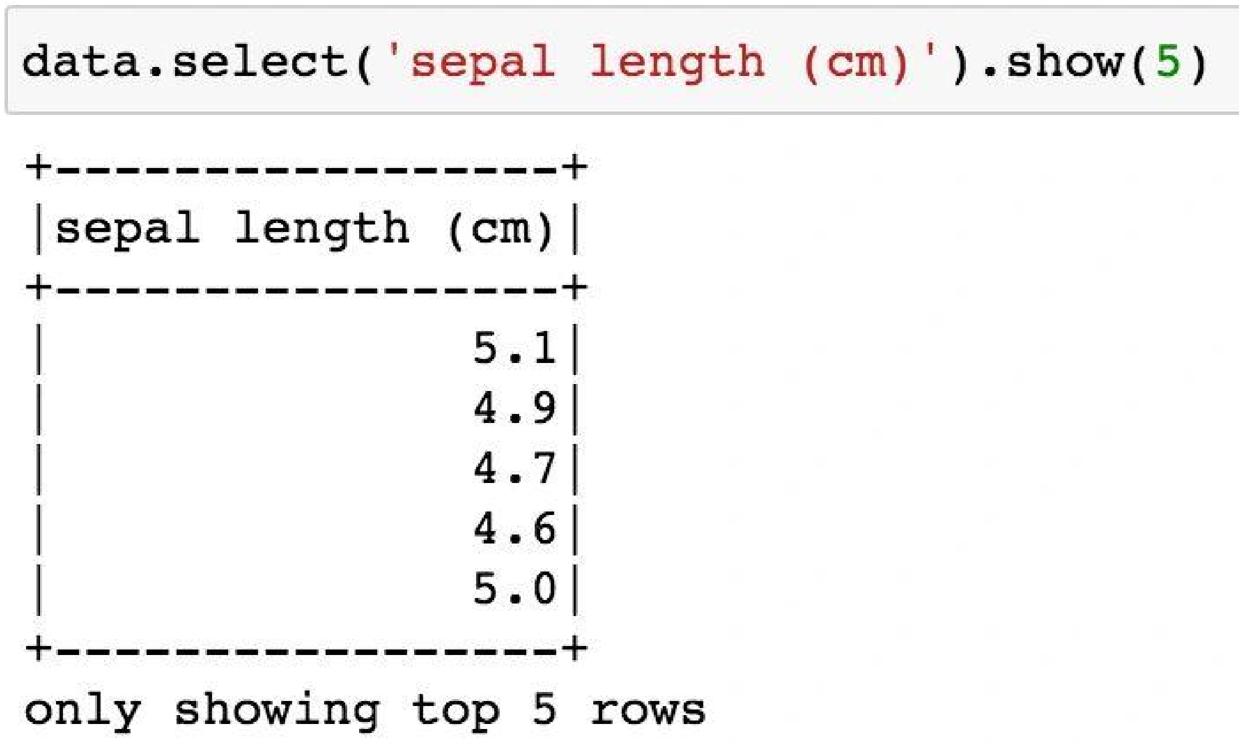
Column selection


```
data.select(data.columns[:2]).show(5)
```

```
+-----------------+----------------+
|sepal length (cm)|sepal width (cm)|
+-----------------+----------------+
|              5.1|             3.5|
|              4.9|             3.0|
|              4.7|             3.2|
|              4.6|             3.1|
|              5.0|             3.6|
+-----------------+----------------+
only showing top 5 rows
```

Edit Image

Select first N columns: Select the First 2 columns from the data frame.

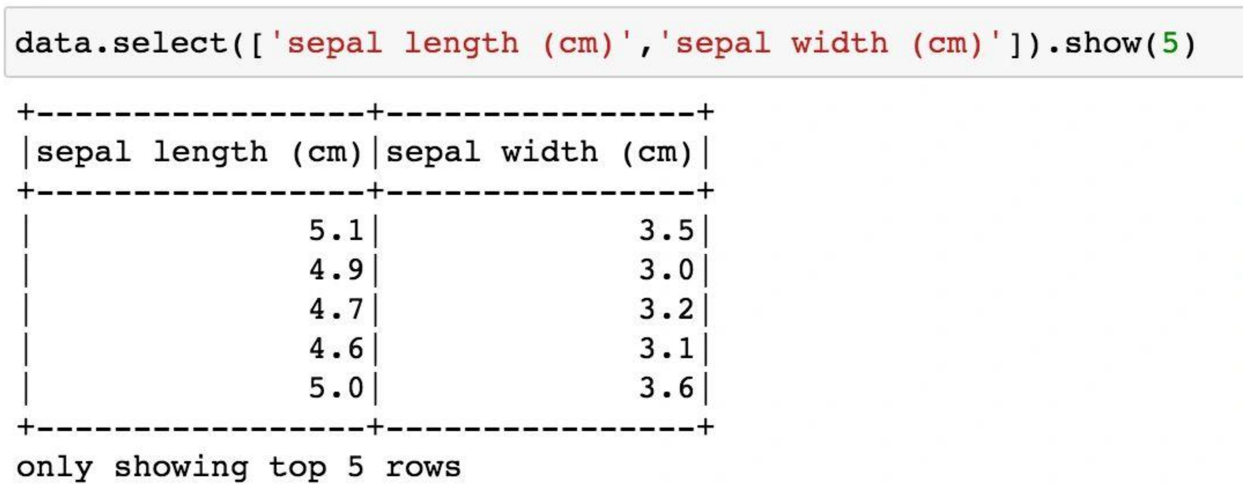


```
data.select('sepal length (cm)').show(5)
```

```
+-----------------+
|sepal length (cm)|
+-----------------+
|              5.1|
|              4.9|
|              4.7|
|              4.6|
|              5.0|
+-----------------+
only showing top 5 rows
```

Edit Image

Single column selection: Select column by using the column name.



```
data.select(['sepal length (cm)','sepal width (cm)']).show(5)
```

```
+-----------------+----------------+
|sepal length (cm)|sepal width (cm)|
+-----------------+----------------+
|              5.1|             3.5|
|              4.9|             3.0|
|              4.7|             3.2|
|              4.6|             3.1|
|              5.0|             3.6|
+-----------------+----------------+
only showing top 5 rows
```

Edit Image

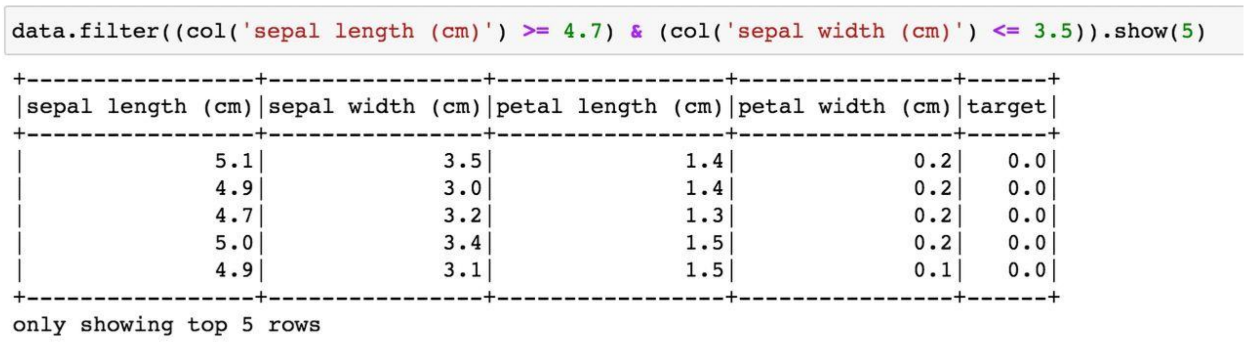
Multiple columns selection: Select columns from a given list of column names.

Row selection

Filter

PySpark filter() will filter the rows from a data frame or RDD based on a condition, where the condition can contain different operators such as OR, NOT, and AND.

Example: Filter records with sepal length with 4.7 and above & sepal width with 3.5 and below.



```
data.filter((col('sepal length (cm)') >= 4.7) & (col('sepal width (cm)') <= 3.5)).show(5)
```

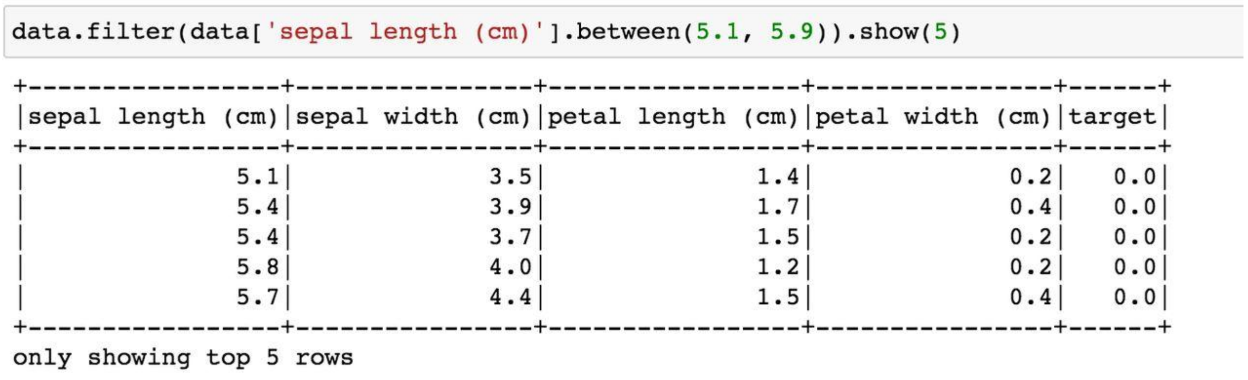
```
+-----------------+----------------+-----------------+----------------+------+
|sepal length (cm)|sepal width (cm)|petal length (cm)|petal width (cm)|target|
+-----------------+----------------+-----------------+----------------+------+
|              5.1|             3.5|              1.4|             0.2|   0.0|
|              4.9|             3.0|              1.4|             0.2|   0.0|
|              4.7|             3.2|              1.3|             0.2|   0.0|
|              5.0|             3.4|              1.5|             0.2|   0.0|
|              4.9|             3.1|              1.5|             0.1|   0.0|
+-----------------+----------------+-----------------+----------------+------+
only showing top 5 rows
```

Edit Image

Between

PySpark between() will result in a boolean value (True or False) based on the condition/expression over the given column.

Example: Filter records with sepal length between 5.1 and 5.9



```
data.filter(data['sepal length (cm)'].between(5.1, 5.9)).show(5)
```

```
+-----------------+----------------+-----------------+----------------+------+
|sepal length (cm)|sepal width (cm)|petal length (cm)|petal width (cm)|target|
+-----------------+----------------+-----------------+----------------+------+
|              5.1|             3.5|              1.4|             0.2|   0.0|
|              5.4|             3.9|              1.7|             0.4|   0.0|
|              5.4|             3.7|              1.5|             0.2|   0.0|
|              5.8|             4.0|              1.2|             0.2|   0.0|
|              5.7|             4.4|              1.5|             0.4|   0.0|
+-----------------+----------------+-----------------+----------------+------+
only showing top 5 rows
```

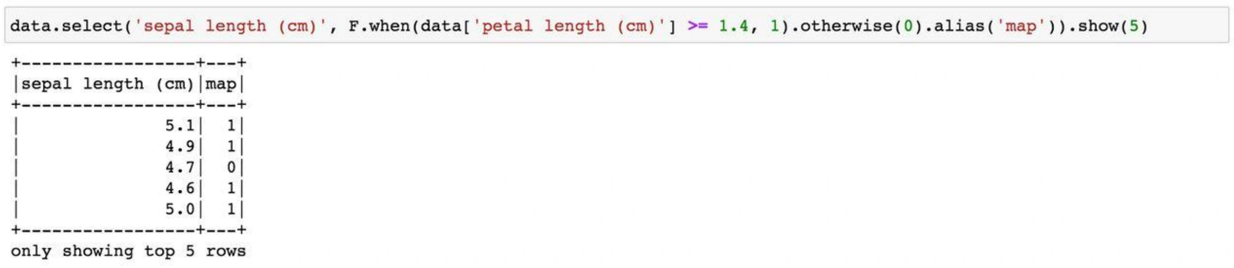
Edit Image

When

PySpark When() can assign a mapping (numeric or string) to a data frame column based on a given condition.

Example:

Suppose there is a requirement to fetch records with sepal length column with petal length 1.4 and above. One of the easy ways to tackle this problem is to use When() functionality by assigning value 1 for records with petal length 1.4 and above, else value 0. The records with map column value 1 will be the required field.



```
data.select('sepal length (cm)', F.when(data['petal length (cm)'] >= 1.4, 1).otherwise(0).alias('map')).show(5)
```

```
+-----------------+---+
|sepal length (cm)|map|
+-----------------+---+
|              5.1|  1|
|              4.9|  1|
|              4.7|  0|
|              4.6|  1|
|              5.0|  1|
+-----------------+---+
only showing top 5 rows
```

Edit Image

<u>Column Manipulation</u>

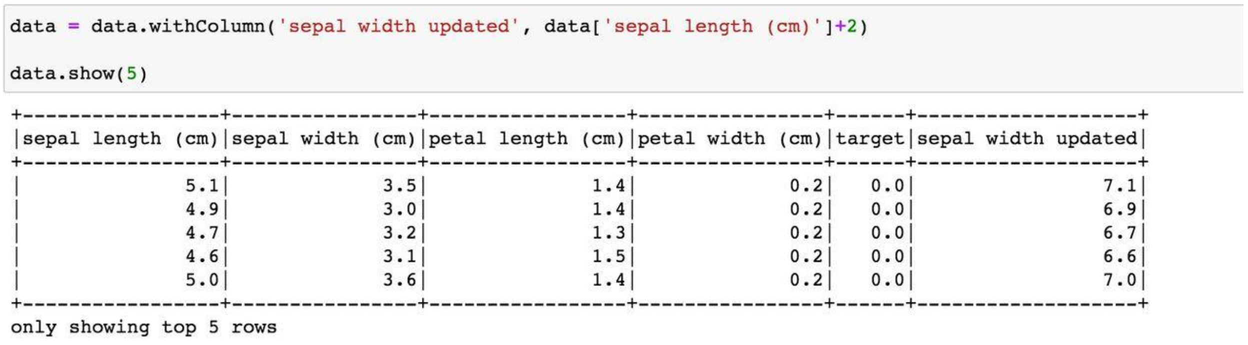
It is possible to perform operations like add, delete, etc over a column of a data frame.

<u>New column addition</u>

A table or data frame can be extended by adding a new column.

<u>Example</u>

Add a new column 'sepal width updated' on the iris data by adding value 2 to the column sepal width.



```
data = data.withColumn('sepal width updated', data['sepal length (cm)']+2)

data.show(5)
```

```
+-----------------+----------------+-----------------+----------------+------+-------------------+
|sepal length (cm)|sepal width (cm)|petal length (cm)|petal width (cm)|target|sepal width updated|
+-----------------+----------------+-----------------+----------------+------+-------------------+
|              5.1|             3.5|              1.4|             0.2|   0.0|                7.1|
|              4.9|             3.0|              1.4|             0.2|   0.0|                6.9|
|              4.7|             3.2|              1.3|             0.2|   0.0|                6.7|
|              4.6|             3.1|              1.5|             0.2|   0.0|                6.6|
|              5.0|             3.6|              1.4|             0.2|   0.0|                7.0|
+-----------------+----------------+-----------------+----------------+------+-------------------+
only showing top 5 rows
```

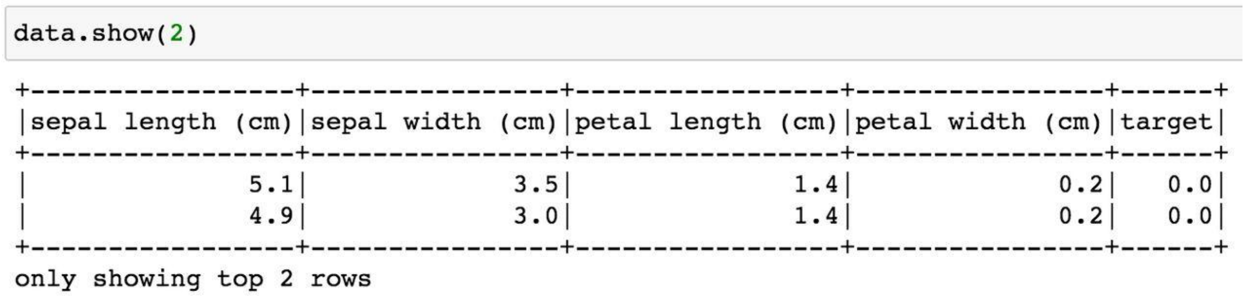
Edit Image

<u>Delete column</u>

A table or data frame can be restricted or shrunk by deleting a column or multiple columns.

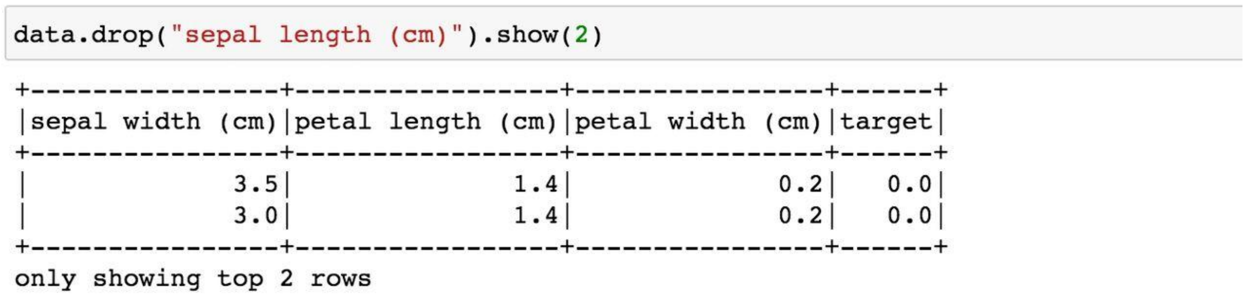
<u>Example</u>

Remove the column 'sepal length (cm)' from the iris data.



```
data.show(2)
```

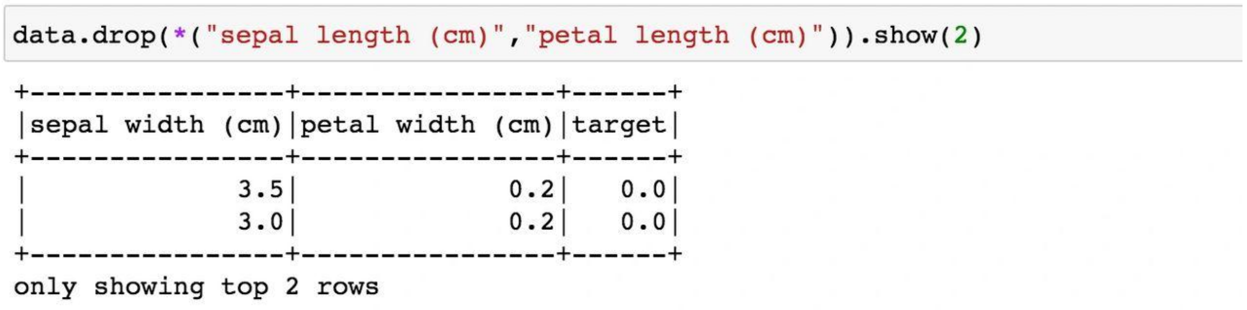
```
+-----------------+----------------+-----------------+----------------+------+
|sepal length (cm)|sepal width (cm)|petal length (cm)|petal width (cm)|target|
+-----------------+----------------+-----------------+----------------+------+
|              5.1|             3.5|              1.4|             0.2|   0.0|
|              4.9|             3.0|              1.4|             0.2|   0.0|
+-----------------+----------------+-----------------+----------------+------+
only showing top 2 rows
```



```
data.drop("sepal length (cm)").show(2)
```

```
+----------------+-----------------+----------------+------+
|sepal width (cm)|petal length (cm)|petal width (cm)|target|
+----------------+-----------------+----------------+------+
|             3.5|              1.4|             0.2|   0.0|
|             3.0|              1.4|             0.2|   0.0|
+----------------+-----------------+----------------+------+
only showing top 2 rows
```

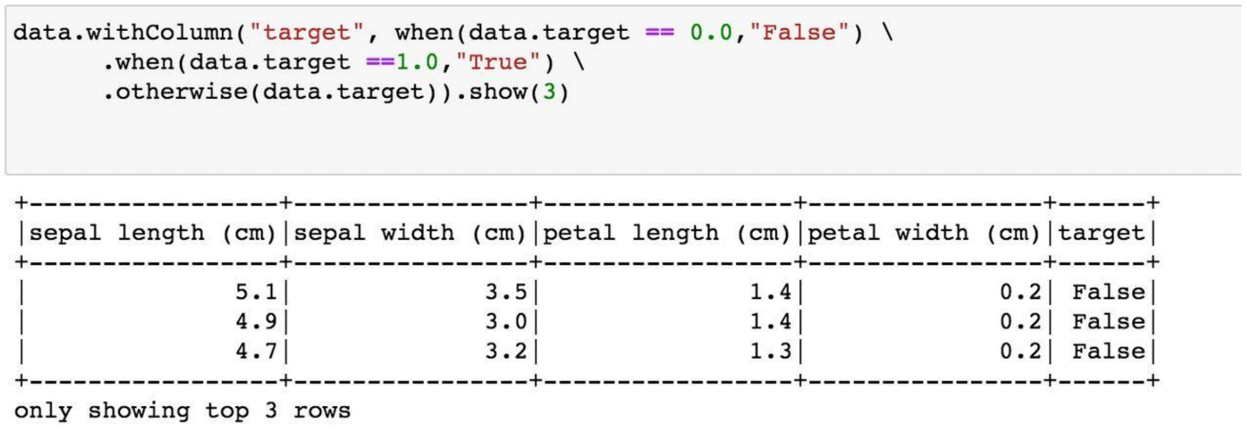
Edit Image
Remove the columns 'sepal length (cm)', 'petal length (cm)' from the iris data.



```
data.drop(*("sepal length (cm)","petal length (cm)")).show(2)
```

```
+----------------+----------------+------+
|sepal width (cm)|petal width (cm)|target|
+----------------+----------------+------+
|             3.5|             0.2|   0.0|
|             3.0|             0.2|   0.0|
+----------------+----------------+------+
only showing top 2 rows
```

Edit Image
Update column values
A column can be updated by numeric or string values as per the requirements.
Example
Update the target column of the iris data by replacing the value 0.0 by 'False' and 1.0 by 'True'.



```python
data.withColumn("target", when(data.target == 0.0,"False") \
      .when(data.target ==1.0,"True") \
      .otherwise(data.target)).show(3)
```

```
+-----------------+----------------+-----------------+----------------+------+
|sepal length (cm)|sepal width (cm)|petal length (cm)|petal width (cm)|target|
+-----------------+----------------+-----------------+----------------+------+
|              5.1|             3.5|              1.4|             0.2| False|
|              4.9|             3.0|              1.4|             0.2| False|
|              4.7|             3.2|              1.3|             0.2| False|
+-----------------+----------------+-----------------+----------------+------+
only showing top 3 rows
```

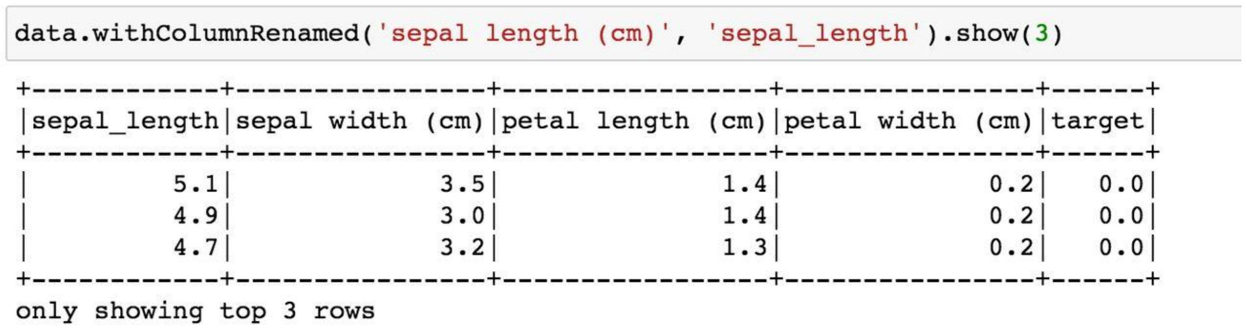
Edit Image

Update column name

Column names can be replaced by using the function withColumnRenamed().

Example

Replace the column name 'sepal length (cm)' by 'sepal_length'.



```python
data.withColumnRenamed('sepal length (cm)', 'sepal_length').show(3)
```

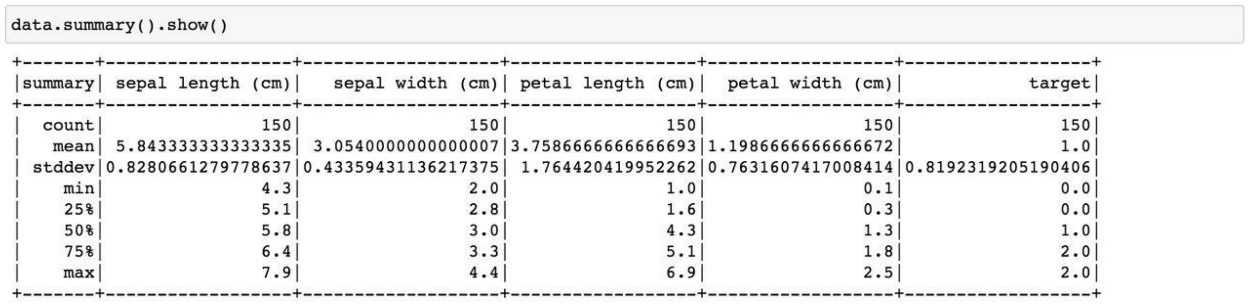
```
+------------+----------------+-----------------+----------------+------+
|sepal_length|sepal width (cm)|petal length (cm)|petal width (cm)|target|
+------------+----------------+-----------------+----------------+------+
|         5.1|             3.5|              1.4|             0.2|   0.0|
|         4.9|             3.0|              1.4|             0.2|   0.0|
|         4.7|             3.2|              1.3|             0.2|   0.0|
+------------+----------------+-----------------+----------------+------+
only showing top 3 rows
```

Edit Image

Summary statistics & Data Aggregations

Summary statistics

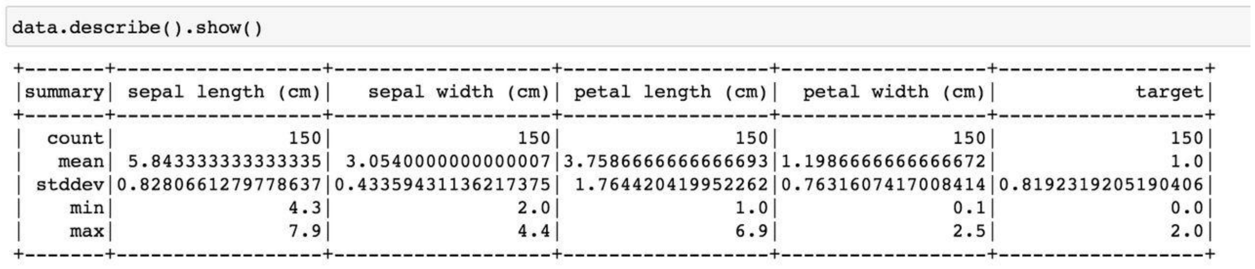
Summary statistics provides a quick overall view of the data set through basic statistics like count, mean, standard deviation, minimum, maximum, and percentiles, etc. It can be obtained by using describe function – describe() and summary(). It is the easiest way to understand the data distribution.



```python
data.summary().show()
```

```
+-------+------------------+-------------------+------------------+------------------+------------------+
|summary| sepal length (cm)|   sepal width (cm)| petal length (cm)|  petal width (cm)|            target|
+-------+------------------+-------------------+------------------+------------------+------------------+
|  count|               150|                150|               150|               150|               150|
|   mean| 5.843333333333335| 3.0540000000000007|3.7586666666666693|1.1986666666666672|               1.0|
| stddev|0.8280661279778637|0.43359431136217375| 1.764420419952262|0.7631607417008414|0.8192319205190406|
|    min|               4.3|                2.0|               1.0|               0.1|               0.0|
|    25%|               5.1|                2.8|               1.6|               0.3|               0.0|
|    50%|               5.8|                3.0|               4.3|               1.3|               1.0|
|    75%|               6.4|                3.3|               5.1|               1.8|               2.0|
|    max|               7.9|                4.4|               6.9|               2.5|               2.0|
+-------+------------------+-------------------+------------------+------------------+------------------+
```

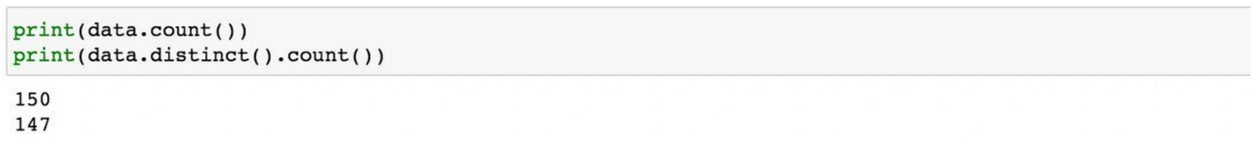
Edit Image



```
data.describe().show()
```

```
+-------+------------------+-------------------+------------------+------------------+------------------+
|summary| sepal length (cm)|   sepal width (cm)| petal length (cm)|  petal width (cm)|            target|
+-------+------------------+-------------------+------------------+------------------+------------------+
|  count|               150|                150|               150|               150|               150|
|   mean| 5.843333333333335| 3.0540000000000007|3.7586666666666693|1.1986666666666672|               1.0|
| stddev|0.8280661279778637|0.43359431136217375| 1.764420419952262|0.7631607417008414|0.8192319205190406|
|    min|               4.3|                2.0|               1.0|               0.1|               0.0|
|    max|               7.9|                4.4|               6.9|               2.5|               2.0|
+-------+------------------+-------------------+------------------+------------------+------------------+
```

Edit Image



```
print(data.count())
print(data.distinct().count())
```

```
150
147
```

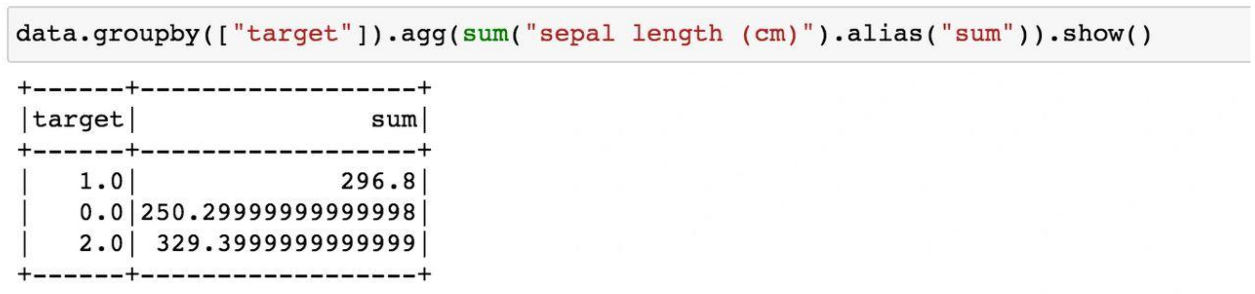
Edit Image

<u>Data Aggregation</u>

Function groupby() has significant role in giving insight of the data through performing different mathematical computations like sum(), mean(), min(), max() etc. The function splits the data into separate groups and will perform computations over these groups. It is possible to specify one or more columns for the grouping and the resulting data frame will consist of the unique value of the grouped columns with the aggregated operation result.

<u>Example</u>

1. Compute sepal length sum for each unique value in the target column.

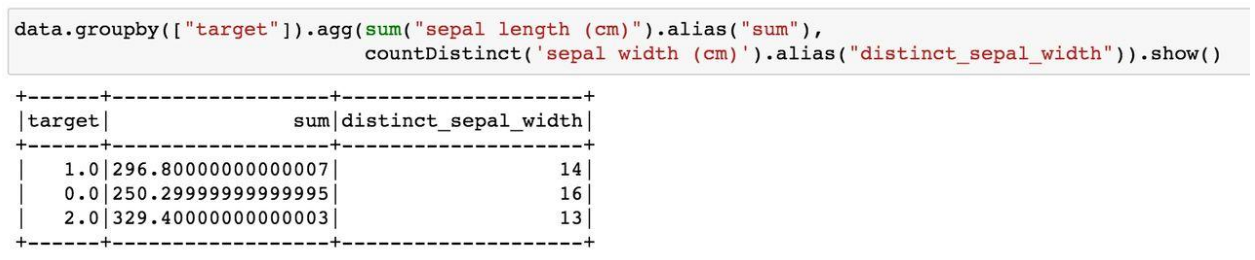


```
data.groupby(["target"]).agg(sum("sepal length (cm)").alias("sum")).show()
```

```
+------+------------------+
|target|               sum|
+------+------------------+
|   1.0|             296.8|
|   0.0|250.29999999999998|
|   2.0| 329.3999999999999|
+------+------------------+
```

Edit Image

2. Along with the sepal length sum, calculate how many distinct sepal widths are there for each unique value in the target column.



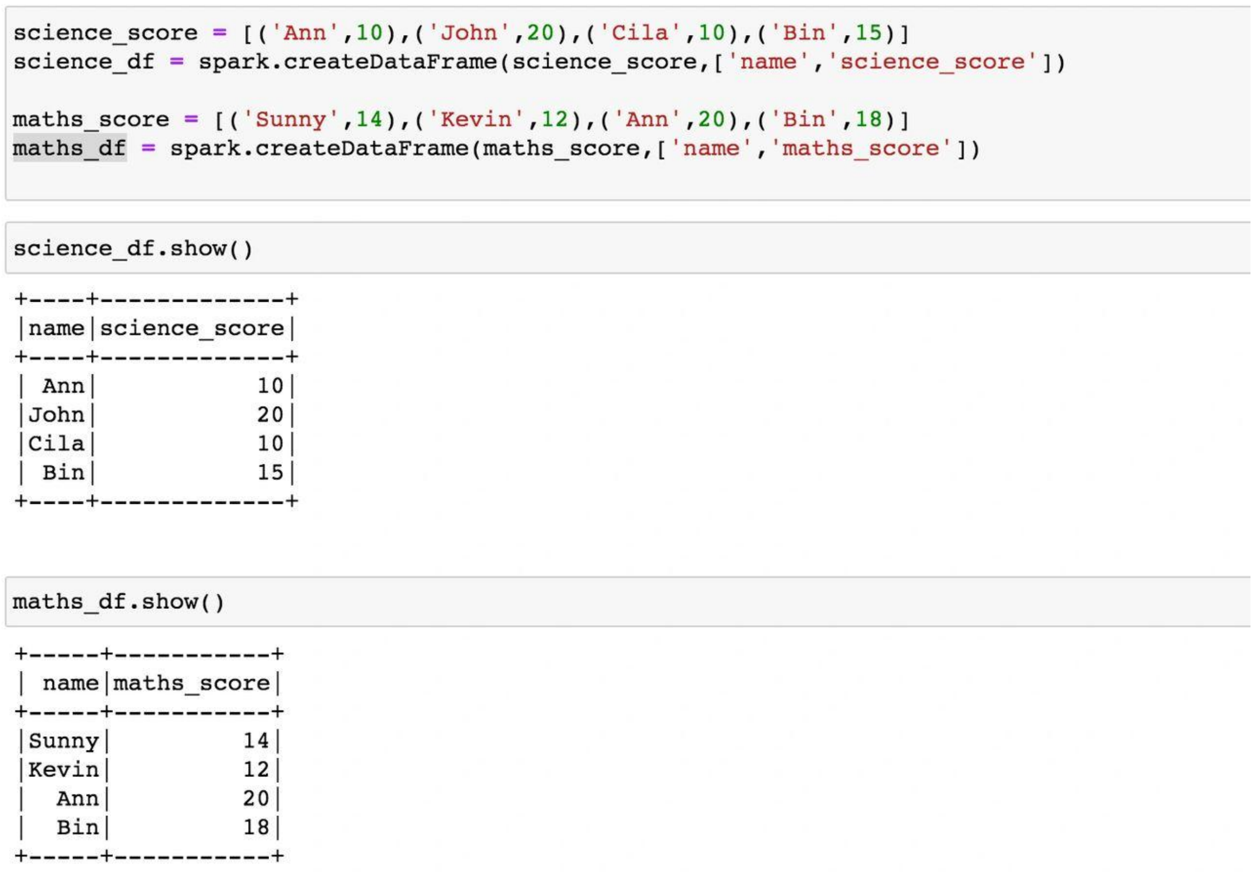
```
data.groupby(["target"]).agg(sum("sepal length (cm)").alias("sum"),
                             countDistinct('sepal width (cm)').alias("distinct_sepal_width")).show()
```

```
+------+------------------+--------------------+
|target|               sum|distinct_sepal_width|
+------+------------------+--------------------+
|   1.0|296.80000000000007|                  14|
|   0.0|250.29999999999995|                  16|
|   2.0|329.40000000000003|                  13|
+------+------------------+--------------------+
```

Edit Image

As we discussed in the above example groupby() can perform different computations over multiple columns effectively in a short time.

<u>Data frame joins</u>
Before discussing data frame joins in detail, let's create two data frames having student name with their score across the subjects Science and Maths.
Lists science_score and maths_score contains (students, score) tuples w.r.s to the subjects. science_df and maths_df are the data frames created based on the respective lists. One of the methods to create a data frame in PySpark is discussed below.



```
science_score = [('Ann',10),('John',20),('Cila',10),('Bin',15)]
science_df = spark.createDataFrame(science_score,['name','science_score'])

maths_score = [('Sunny',14),('Kevin',12),('Ann',20),('Bin',18)]
maths_df = spark.createDataFrame(maths_score,['name','maths_score'])
```

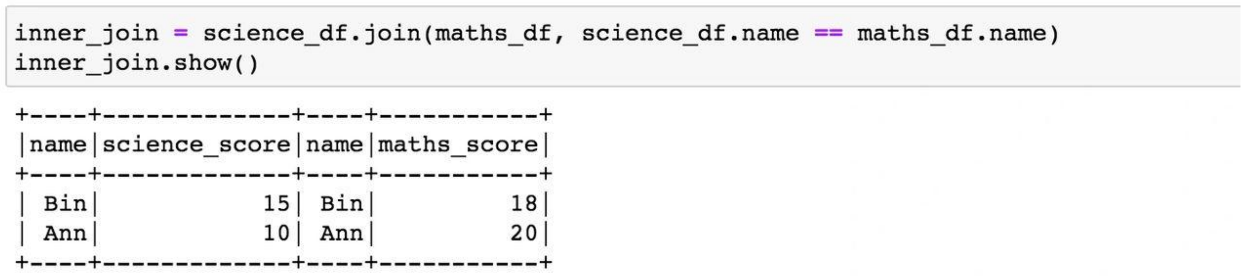
```
science_df.show()
```

```
+----+-------------+
|name|science_score|
+----+-------------+
| Ann|           10|
|John|           20|
|Cila|           10|
| Bin|           15|
+----+-------------+
```

```
maths_df.show()
```

```
+-----+-----------+
| name|maths_score|
+-----+-----------+
|Sunny|         14|
|Kevin|         12|
|  Ann|         20|
|  Bin|         18|
+-----+-----------+
```

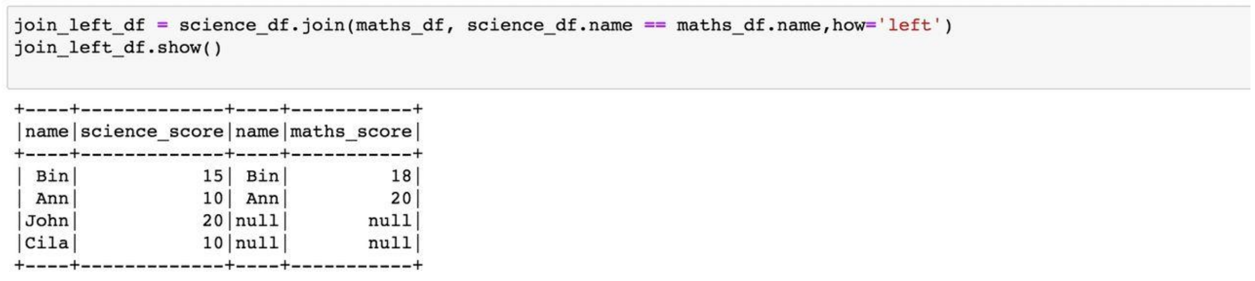
Edit Image
Following are the 4 major types of join operations common in use.
<u>Inner Join</u>
Return only the rows in which the left table has matching keys in the right table.



```
inner_join = science_df.join(maths_df, science_df.name == maths_df.name)
inner_join.show()
```

```
+----+-------------+----+-----------+
|name|science_score|name|maths_score|
+----+-------------+----+-----------+
| Bin|           15| Bin|         18|
| Ann|           10| Ann|         20|
+----+-------------+----+-----------+
```

Edit Image
<u>Left Join</u>
Return all rows from the left table and any rows with matching keys from the right table.



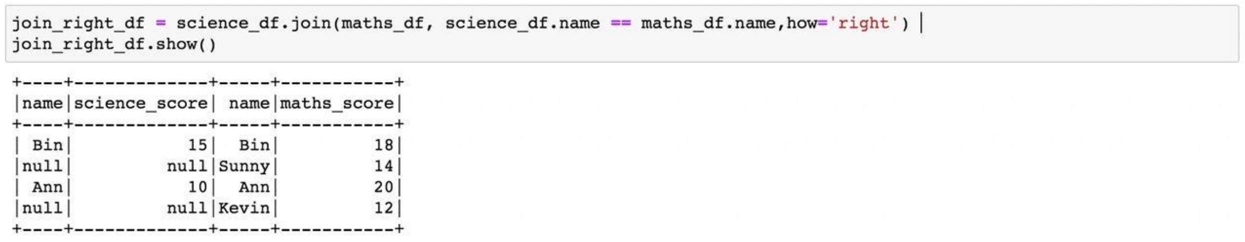
```
join_left_df = science_df.join(maths_df, science_df.name == maths_df.name,how='left')
join_left_df.show()
```

```
+----+-------------+----+-----------+
|name|science_score|name|maths_score|
+----+-------------+----+-----------+
| Bin|           15| Bin|         18|
| Ann|           10| Ann|         20|
|John|           20|null|       null|
|Cila|           10|null|       null|
+----+-------------+----+-----------+
```

Edit Image

Right Join

Return all rows from the right table and any rows with matching keys from the left table.



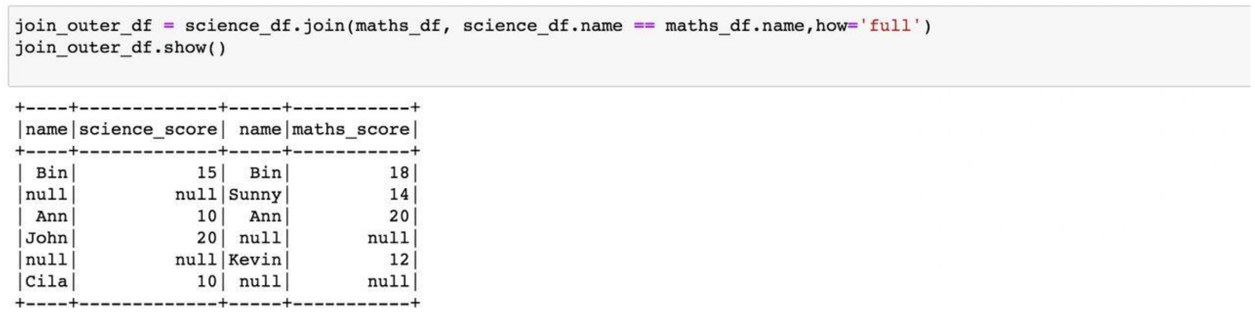
```
join_right_df = science_df.join(maths_df, science_df.name == maths_df.name,how='right')
join_right_df.show()
```

```
+----+-------------+-----+-----------+
|name|science_score| name|maths_score|
+----+-------------+-----+-----------+
| Bin|           15|  Bin|         18|
|null|         null|Sunny|         14|
| Ann|           10|  Ann|         20|
|null|         null|Kevin|         12|
+----+-------------+-----+-----------+
```

Edit Image

Outer Join

Returns all rows from both tables, join the left records with matching keys in the right table.



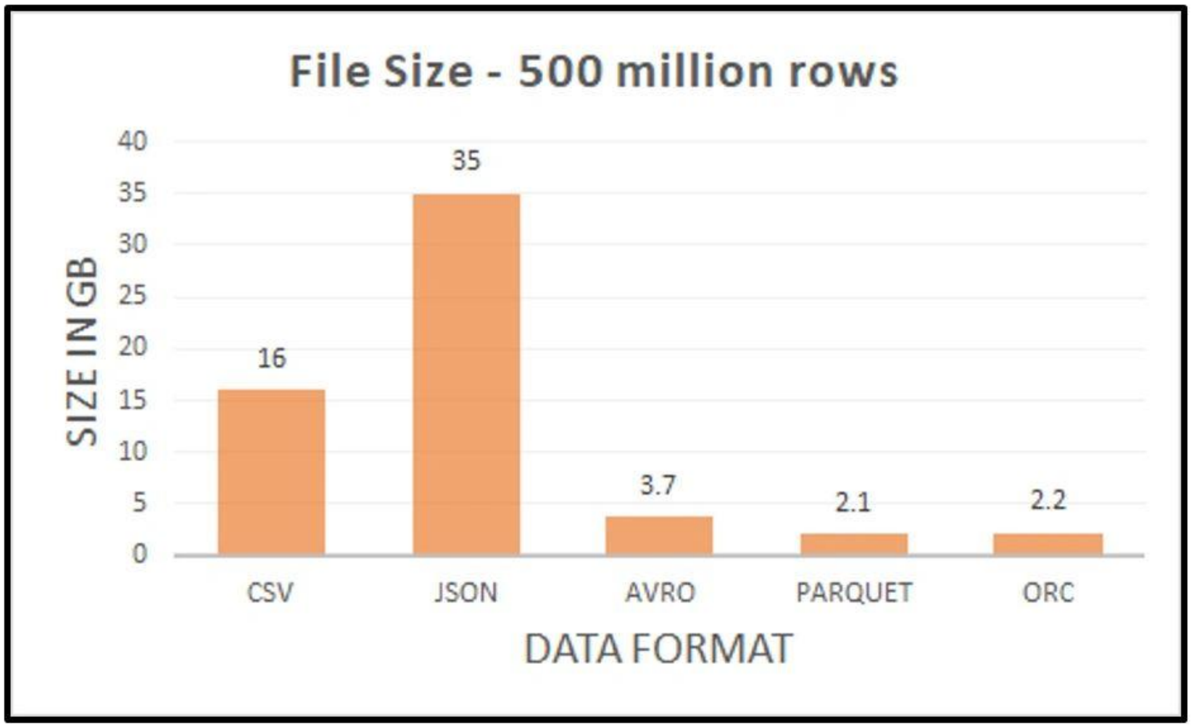
```
join_outer_df = science_df.join(maths_df, science_df.name == maths_df.name,how='full')
join_outer_df.show()
```

```
+----+-------------+-----+-----------+
|name|science_score| name|maths_score|
+----+-------------+-----+-----------+
| Bin|           15|  Bin|         18|
|null|         null|Sunny|         14|
| Ann|           10|  Ann|         20|
|John|           20| null|       null|
|null|         null|Kevin|         12|
|Cila|           10| null|       null|
+----+-------------+-----+-----------+
```

Edit Image

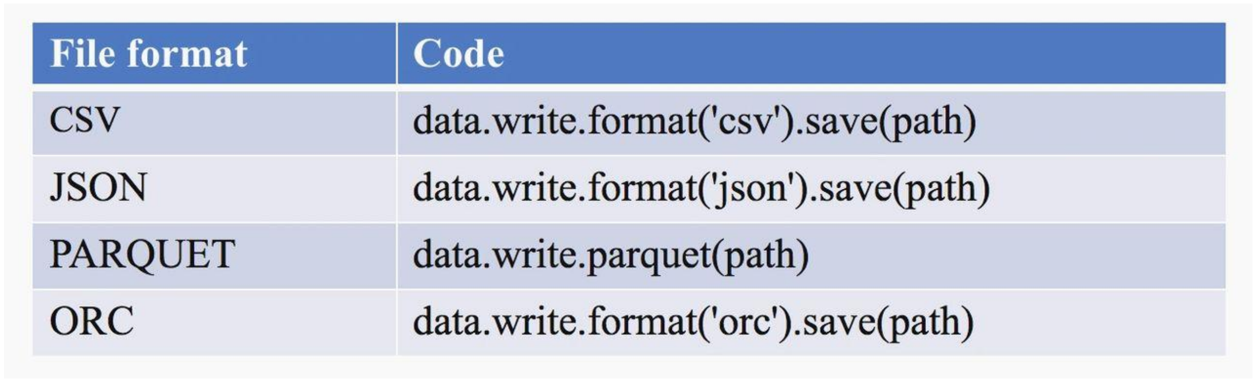
Save or Write data frames in different file formats

It is possible to store the data in different formats such as CSV, JSON, PARQUET and ORC, etc. Saving/writing space and time for each file format are different across these formats.



Edit Image
File format order based on the writing / saving space

Let's see how to save the data in different file formats.



File format	Code
CSV	data.write.format('csv').save(path)
JSON	data.write.format('json').save(path)
PARQUET	data.write.parquet(path)
ORC	data.write.format('orc').save(path)

Edit Image

<u>Conclusion</u>

PySpark has a significant role in analyzing Big data efficiently, which actually reflects in solving many business problems. Python is the best option to play with different data but has limitations if the data size is large. So, while dealing with a large amount of data, the python programmers can make the best use of this tool to complete the need.