How to Find Unique Values in a Column

The easiest way to obtain a list of unique values in a PySpark DataFrame column is to use the **distinct** function.

This tutorial provides several examples of how to use this function with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11],
        ['A', 'East', 8],
        ['A', 'East', 10],
        ['B', 'West', 6],
        ['B', 'West', 6],
        ['C', 'East', 5]]

#define column names
columns = ['team', 'conference', 'points']

#create DataFrame using data and column names
df = spark.createDataFrame(data, columns)

#view DataFrame
df.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
|   A|      East|    11|
|   A|      East|     8|
|   A|      East|    10|
|   B|      West|     6|
|   B|      West|     6|
|   C|      East|     5|
+----+----------+------+
```

Example 1: Find Unique Values in a Column

We can use the following syntax to find the unique values in the **team** column of the DataFrame:

```
df.select('team').distinct().show()
```

```
+----+
|team|
+----+
|   A|
|   B|
|   C|
+----+
```

We can see that the unique values in the **team** column are **A**, **B** and **C**.

Example 2: Find and Sort Unique Values in a Column

Suppose we used the following syntax to find the unique values in the **points** column:

```
df.select('points').distinct().show()

+------+
|points|
+------+
|    11|
|     8|
|    10|
|     6|
|     5|
+------+
```

The output displays the unique values but they aren't sorted in any way.

If we'd like, we can use the following syntax to find the unique values in the **points** column and return them sorted in **ascending order**:

```
#find unique values in points column
df_points = df.select('points').distinct()

#display unique values in ascending order
df_points.orderBy('points').show()

+------+
|points|
+------+
|     5|
|     6|
|     8|
|    10|
|    11|
+------+
```

We can also use the argument **ascending=False** to return the unique values in **descending order** instead:

```
#find unique values in points column
df_points = df.select('points').distinct()

#display unique values in descending order
df_points.orderBy('points', ascending=False).show()

+------+
|points|
+------+
|    11|
|    10|
|     8|
|     6|
|     5|
+------+
```

Example 3: Find and Count Unique Values in a Column

The following code shows how to find and count the occurrence of unique values in the **team** column of the DataFrame:

```
df.groupBy('team').count().show()

+----+-----+
|team|count|
+----+-----+
|   A|    3|
|   B|    2|
|   C|    1|
+----+-----+
```

From the output we can see the three unique values (**A**, **B**, **C**) along with the number of times each unique value occurs.

How to Select Rows by Index in DataFrame

By default, a PySpark DataFrame does not have a built-in index.

However, it's easy to add an index column which you can then use to select rows in the DataFrame based on their index value.

The following example shows how to do so in practice.

Example: Select Rows by Index in PySpark DataFrame

Suppose we create the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11],
        ['A', 'East', 8],
        ['A', 'East', 10],
        ['B', 'West', 6],
        ['B', 'West', 6],
        ['C', 'East', 5]]

#define column names
columns = ['team', 'conference', 'points']

#create DataFrame using data and column names
df = spark.createDataFrame(data, columns)

#view DataFrame
df.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
|   A|      East|    11|
|   A|      East|     8|
|   A|      East|    10|
|   B|      West|     6|
|   B|      West|     6|
|   C|      East|     5|
+----+----------+------+
```

We can use the following syntax to add a column called **id** that ranges from 1 to the last row in the DataFrame:

```
from pyspark.sql.functions import row_number,lit
from pyspark.sql.window import Window

#add column called 'id' that contains row numbers from 1 to n
w = Window().orderBy(lit('A'))
df = df.withColumn('id', row_number().over(w))

#view updated DataFrame
```

```
df.show()

+----+----------+------+---+
|team|conference|points| id|
+----+----------+------+---+
|   A|      East|    11|  1|
|   A|      East|     8|  2|
|   A|      East|    10|  3|
|   B|      West|     6|  4|
|   B|      West|     6|  5|
|   C|      East|     5|  6|
+----+----------+------+---+
```

Now we can use the **where** function along with the **between** function to select all rows between index values 2 and 5:

```
from pyspark.sql.functions import col

#select all rows between index values 2 and 5
df.where(col('id').between(2, 5)).show()

+----+----------+------+---+
|team|conference|points| id|
+----+----------+------+---+
|   A|      East|     8|  2|
|   A|      East|    10|  3|
|   B|      West|     6|  4|
|   B|      West|     6|  5|
+----+----------+------+---+
```

The output displays all rows in the DataFrame between index values 2 and 5.

If we'd like, we can instead use the **filter** and **isin** functions to select specific rows in a list:

```
#find unique values in points column
df.filter(df.id.isin(1,5,6)).show()

+----+----------+------+---+
|team|conference|points| id|
+----+----------+------+---+
|   A|      East|    11|  1|
|   B|      West|     6|  5|
|   C|      East|     5|  6|
+----+----------+------+---+
```

The output displays the rows in the DataFrame in index positions **1**, **5** and **6**.

How to Select Columns by Index in DataFrame

You can use the following methods to select columns by index in a PySpark DataFrame:

Method 1: Select Specific Column by Index

```
#select first column in DataFrame
df.select(df.columns[0]).show()
```

Method 2: Select All Columns Except Specific One by Index

```
#select all columns except first column in DataFrame
df.drop(df.columns[0]).show()
```

Method 3: Select Range of Columns by Index

```
#select all columns between index 0 and 2, not including 2
df.select(df.columns[0:2]).show()
```

The following examples show how to use each of these methods in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11],
        ['A', 'East', 8],
        ['A', 'East', 10],
        ['B', 'West', 6],
        ['B', 'West', 6],
        ['C', 'East', 5]]

#define column names
columns = ['team', 'conference', 'points']

#create DataFrame using data and column names
df = spark.createDataFrame(data, columns)

#view DataFrame
df.show()

+----+----------+------+
```

```
|team|conference|points|
+----+----------+------+
|   A|      East|    11|
|   A|      East|     8|
|   A|      East|    10|
|   B|      West|     6|
|   B|      West|     6|
|   C|      East|     5|
+----+----------+------+
```

Example 1: Select Specific Column by Index

We can use the following syntax to select only the first column in the DataFrame:

```
#select first column in DataFrame
df.select(df.columns[0]).show()

+----+
|team|
+----+
|   A|
|   A|
|   A|
|   B|
|   B|
|   C|
+----+
```

Notice that only the first column (the **team** column) has been selected from the DataFrame.

Example 2: Select All Columns Except Specific One by Index

We can use the following syntax to select all columns in the DataFrame *except* for the first column:

```
#select all columns except first column in DataFrame
df.drop(df.columns[0]).show()

+----------+------+
|conference|points|
+----------+------+
|      East|    11|
|      East|     8|
|      East|    10|
|      West|     6|
|      West|     6|
```

```
|      East|     5|
+----------+------+
```

Notice that all columns except the first column (the **team** column) have been selected from the DataFrame.

Example 3: Select Range of Columns by Index

We can use the following syntax to select all columns in the DataFrame in the range of 0 to 2 (not including 2):

```
#select all columns between index 0 and 2, not including 2
df.select(df.columns[0:2]).show()

+----+----------+
|team|conference|
+----+----------+
|   A|      East|
|   A|      East|
|   A|      East|
|   B|      West|
|   B|      West|
|   C|      East|
+----+----------+
```

Notice that all columns in the range of 0 to 2 (not including 2) have been selected from the DataFrame.

How to Select Rows Based on Column Values

You can use the following methods to select rows based on column values in a PySpark DataFrame:

Method 1: Select Rows where Column is Equal to Specific Value

```
#select rows where 'team' column is equal to 'B'
df.where(df.team=='B').show()
```

Method 2: Select Rows where Column Value is in List of Values

```
#select rows where 'team' column is equal to 'A' or 'B'
df.filter(df.team.isin('A','B')).show()
```

Method 3: Select Rows Based on Multiple Column Conditions

```
#select rows where 'team' column is 'A' and 'points' column is
greater than 9
df.where((df.team=='A') & (df.points>9)).show()
```

The following examples show how to use each of these methods in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11],
        ['A', 'East', 8],
        ['A', 'East', 10],
        ['B', 'West', 6],
        ['B', 'West', 6],
        ['C', 'East', 5]]

#define column names
columns = ['team', 'conference', 'points']

#create DataFrame using data and column names
df = spark.createDataFrame(data, columns)

#view DataFrame
df.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
|   A|      East|    11|
|   A|      East|     8|
|   A|      East|    10|
|   B|      West|     6|
|   B|      West|     6|
|   C|      East|     5|
+----+----------+------+
```

Example 1: Select Rows where Column is Equal to Specific Value

We can use the following syntax to select only the rows where the **team** column is equal to **B**:

```
#select rows where 'team' column is equal to 'B'
df.where(df.team=='B').show()

+----+----------+------+
|team|conference|points|
```

```
+----+----------+------+
|   B|      West|     6|
|   B|      West|     6|
+----+----------+------+
```

Notice that only the rows where the **team** column is equal to **B** are returned.

Example 2: Select Rows where Column Value is in List of Values

We can use the following syntax to select only the rows where the **team** column is equal to **A** or **B**:

```
#select rows where 'team' column is equal to 'A' or 'B'
df.filter(df.team.isin('A','B')).show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
|   A|      East|    11|
|   A|      East|     8|
|   A|      East|    10|
|   B|      West|     6|
|   B|      West|     6|
+----+----------+------+
```

Notice that only the rows where the **team** column is equal to either **A** or **B** are returned.

Example 3: Select Rows Based on Multiple Column Conditions

We can use the following syntax to select only the rows where the **team** column is equal to **A** *and* the **points** column is greater than **9**:

```
#select rows where 'team' column is 'A' and 'points' column is
greater than 9
df.where((df.team=='A') & (df.points>9)).show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
|   A|      East|    11|
|   A|      East|    10|
+----+----------+------+
```

Notice that only the two rows that met both conditions are returned.

How to Keep Certain Columns in PySpark

You can use the following methods to only keep certain columns in a PySpark DataFrame:

Method 1: Specify Columns to Keep

```
from pyspark.sql.functions import col

#only keep columns 'col1' and 'col2'
df.select(col('col1'), col('col2')).show()
```

Method 2: Specify Columns to Drop

```
from pyspark.sql.functions import col

#drop columns 'col3' and 'col4'
df.drop(col('col3'), col('col4')).show()
```

The following examples show how to use each method with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
```

```
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Specify Columns to Keep

The following code shows how to define a new DataFrame that only keeps the **team** and **points** columns:

```
from pyspark.sql.functions import col

#create new DataFrame and only keep 'team' and 'points' columns
df.select(col('team'), col('points')).show()

+----+------+
|team|points|
+----+------+
|   A|    11|
|   A|     8|
|   A|    10|
|   B|     6|
|   B|     6|
|   C|     5|
+----+------+
```

Notice that the resulting DataFrame only keeps the two columns that we specified.

Example 2: Specify Columns to Drop

The following code shows how to define a new DataFrame that drops the **conference** and **assists** columns from the original DataFrame:

```
from pyspark.sql.functions import col

#create new DataFrame that drops 'conference' and 'assists' columns
df.drop(col('conference'), col('assists')).show()

+----+------+
|team|points|
+----+------+
|   A|    11|
|   A|     8|
|   A|    10|
|   B|     6|
|   B|     6|
```

```
|   C|     5|
+----+------+
```

Notice that the resulting DataFrame drops the **conference** and **assists** columns from the original DataFrame and keeps the remaining columns.

How to Select Multiple Columns in PySpark

There are three common ways to select multiple columns in a PySpark DataFrame:

Method 1: Select Multiple Columns by Name

```
#select 'team' and 'points' columns
df.select('team', 'points').show()
```

Method 2: Select Multiple Columns Based on List

```
#define list of columns to select
select_cols = ['team', 'points']

#select all columns in list
df.select(*select_cols).show()
```

Method 3: Select Multiple Columns Based on Index Range

```
#select all columns between index 0 and 2 ( not including 2)
df.select(df.columns[0:2]).show()
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']
```

```
#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Select Multiple Columns by Name

We can use the following syntax to select the **team** and **points** columns of the DataFrame:

```
#select 'team' and 'points' columns
df.select('team', 'points').show()

+----+------+
|team|points|
+----+------+
|   A|    11|
|   A|     8|
|   A|    10|
|   B|     6|
|   B|     6|
|   C|     5|
+----+------+
```

Notice that the resulting DataFrame only contains the **team** and **points** columns, just as we specified.

Example 2: Select Multiple Columns Based on List

We can use the following syntax to specify a list of column names and then select all columns in the DataFrame that belong to the list:

```
#define list of columns to select
select_cols = ['team', 'points']

#select all columns in list
```

```
df.select(*select_cols).show()

+----+------+
|team|points|
+----+------+
|   A|    11|
|   A|     8|
|   A|    10|
|   B|     6|
|   B|     6|
|   C|     5|
+----+------+
```

Notice that the resulting DataFrame only contains the column names that we specified in the list.

Example 3: Select Multiple Columns Based on Index Range

We can use the following syntax to specify a list of column names and then select all columns in the DataFrame that belong to the list:

```
#select all columns between index positions 0 and 2 ( not including
2)
df.select(df.columns[0:2]).show()

+----+----------+
|team|conference|
+----+----------+
|   A|      East|
|   A|      East|
|   A|      East|
|   B|      West|
|   B|      West|
|   C|      East|
+----+----------+
```

Notice that the resulting DataFrame only contains the columns in index positions 0 and 1.

How to Do a Left Join in PySpark (With Example)

You can use the following basic syntax to perform a left join in PySpark:

```
df_joined = df1.join(df2, on=['team'], how='left').show()
```

This particular example will perform a left join using the DataFrames named **df1** and **df2** by joining on the column named **team**.

All rows from **df1** will be returned in the final DataFrame but only the rows from **df2** that have a matching value in the **team** column will be returned.

The following example shows how to use this syntax in practice.

Example: How to Do a Left Join in PySpark

Suppose we have the following DataFrame named **df1**:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data1 = [['Mavs', 11],
       ['Hawks', 25],
       ['Nets', 32],
       ['Kings', 15],
       ['Warriors', 22],
       ['Suns', 17]]

#define column names
columns1 = ['team', 'points']

#create dataframe using data and column names
df1 = spark.createDataFrame(data1, columns1)

#view dataframe
df1.show()

+--------+------+
|    team|points|
+--------+------+
|    Mavs|    11|
|   Hawks|    25|
|    Nets|    32|
|   Kings|    15|
|Warriors|    22|
|    Suns|    17|
+--------+------+
```

And suppose we have another DataFrame named **df2**:

```
#define data
data2 = [['Mavs', 4],
       ['Nets', 7],
```

```
       ['Suns', 8],
       ['Grizzlies', 12],
       ['Kings', 7]]

#define column names
columns2 = ['team', 'assists']

#create dataframe using data and column names
df2 = spark.createDataFrame(data2, columns2)

#view dataframe
df2.show()

+---------+-------+
|     team|assists|
+---------+-------+
|     Mavs|      4|
|     Nets|      7|
|     Suns|      8|
|Grizzlies|     12|
|    Kings|      7|
+---------+-------+
```

We can use the following syntax to perform a left join between these two DataFrames by joining on values from the **team** column:

```
#perform left join using 'team' column
df_joined = df1.join(df2, on=['team'], how='left').show()

+--------+------+-------+
|    team|points|assists|
+--------+------+-------+
|    Mavs|    11|      4|
|   Hawks|    25|   null|
|    Nets|    32|      7|
|   Kings|    15|      7|
|Warriors|    22|   null|
|    Suns|    17|      8|
+--------+------+-------+
```

Notice that the resulting DataFrame contains all rows from the left DataFrame (**df1**) but only the rows from the right DataFrame (**df2**) that had a matching value in the **team** column.

Note that if the right DataFrame did not contain a matching team value for any team in the left DataFrame, a value of **null** is used in the **assists** column.

For example, the team name “Hawks” did not exist in **df2**, so this row received a value of **null** in the **assists** column of the final joined DataFrame.

PySpark: How to Do a Left Join on Multiple Columns

You can use the following syntax in PySpark to perform a left join using multiple columns:

```
df_joined = df1.join(df2, on=[df1.col1==df2.col1,
df1.col2==df2.col2], how='left')
```

This particular example will perform a left join using the DataFrames named **df1** and **df2** by joining on the columns named **col1** and **col2**.

All rows from **df1** will be returned in the final DataFrame but only the rows from **df2** that have a matching value in the columns named **col1** and **col2** will be returned.

The following example shows how to use this syntax in practice.

Example: Left Join on Multiple Columns in PySpark

Suppose we have the following DataFrame named **df1**:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data1 = [['A', 'G', 18],
       ['A', 'F', 22],
       ['B', 'F', 19],
       ['B', 'G', 14]]

#define column names
columns1 = ['team', 'pos', 'points']

#create dataframe using data and column names
df1 = spark.createDataFrame(data1, columns1)

#view dataframe
df1.show()

+----+---+------+
|team|pos|points|
+----+---+------+
|   A|  G|    18|
|   A|  F|    22|
```

```
|   B|  F|    19|
|   B|  G|    14|
+----+---+------+
```

And suppose we have another DataFrame named **df2**:

```
#define data
data2 = [['A', 'G', 4],
        ['A', 'F', 9],
        ['B', 'F', 8],
        ['C', 'G', 6],
        ['C', 'F', 5]]

#define column names
columns2 = ['team_name', 'position', 'assists']

#create dataframe using data and column names
df2 = spark.createDataFrame(data2, columns2)

#view dataframe
df2.show()

+---------+--------+-------+
|team_name|position|assists|
+---------+--------+-------+
|        A|       G|      4|
|        A|       F|      9|
|        B|       F|      8|
|        C|       G|      6|
|        C|       F|      5|
+---------+--------+-------+
```

Suppose we would like to perform a left join between these two DataFrames by joining on the following columns:

- Where **team** from **df1** matches **team_name** from **df2**.
- Where **pos** from **df1** matches **position** from **df2**.

We can use the following syntax to do so:

```
#perform left join
df_joined = df1.join(df2, on=[df1.team==df2.team_name,
df1.pos==df2.position], how='left')

#view resulting DataFrame
df_joined.show()

+----+---+------+---------+--------+-------+
```

```
|team|pos|points|team_name|position|assists|
+----+---+------+---------+--------+-------+
|   A|  G|    18|        A|       G|      4|
|   A|  F|    22|        A|       F|      9|
|   B|  F|    19|        B|       F|      8|
|   B|  G|    14|     null|    null|   null|
+----+---+------+---------+--------+-------+
```

Lastly, we can drop the **team_name** and **position** columns from the resulting DataFrame since they're redundant:

```
#drop 'team_name' and 'position' columns from joined DataFrame
df_joined.drop('team_name', 'position').show()

+----+---+------+-------+
|team|pos|points|assists|
+----+---+------+-------+
|   A|  G|    18|      4|
|   A|  F|    22|      9|
|   B|  F|    19|      8|
|   B|  G|    14|   null|
+----+---+------+-------+
```

We have now successfully performed a left join using multiple columns.

Note that all rows from the left DataFrame (**df1**) exist in the joined DataFrame, but only the rows from the right DataFrame (**df2**) that had matching values in both the **team** and **position** columns made it to the final joined DataFrame.

How to Do a Right Join in PySpark

You can use the following basic syntax to perform a right join in PySpark:

```
df_joined = df1.join(df2, on=['team'], how='right ').show()
```

This particular example will perform a right join using the DataFrames named **df1** and **df2** by joining on the column named **team**.

All rows from **df2** will be returned in the final DataFrame but only the rows from **df1** that have a matching value in the **team** column will be returned.

The following example shows how to use this syntax in practice.

Example: How to Do a Right Join in PySpark

Suppose we have the following DataFrame named **df1**:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data1 = [['Mavs', 11],
       ['Hawks', 25],
       ['Nets', 32],
       ['Kings', 15],
       ['Warriors', 22],
       ['Suns', 17]]

#define column names
columns1 = ['team', 'points']

#create dataframe using data and column names
df1 = spark.createDataFrame(data1, columns1)

#view dataframe
df1.show()

+--------+------+
|    team|points|
+--------+------+
|    Mavs|    11|
|   Hawks|    25|
|    Nets|    32|
|   Kings|    15|
|Warriors|    22|
|    Suns|    17|
+--------+------+
```

And suppose we have another DataFrame named **df2**:

```
#define data
data2 = [['Mavs', 4],
       ['Nets', 7],
       ['Suns', 8],
       ['Grizzlies', 12],
       ['Kings', 7]]

#define column names
columns2 = ['team', 'assists']

#create dataframe using data and column names
```

```
df2 = spark.createDataFrame(data2, columns2)

#view dataframe
df2.show()

+---------+-------+
|     team|assists|
+---------+-------+
|     Mavs|      4|
|     Nets|      7|
|     Suns|      8|
|Grizzlies|     12|
|    Kings|      7|
+---------+-------+
```

We can use the following syntax to perform a right join between these two DataFrames by joining on values from the **team** column:

```
#perform right join using 'team' column
df_joined = df1.join(df2, on=['team'], how='right').show()

+---------+------+-------+
|     team|points|assists|
+---------+------+-------+
|     Mavs|    11|      4|
|     Nets|    32|      7|
|     Suns|    17|      8|
|Grizzlies|  null|     12|
|    Kings|    15|      7|
+---------+------+-------+
```

Notice that the resulting DataFrame contains all rows from the right DataFrame (**df2**) but only the rows from the left DataFrame (**df1**) that had a matching value in the **team** column.

Note that if the left DataFrame did not contain a matching team value for any team in the right DataFrame, a value of **null** is used in the **points** column.

For example, the team name "Grizzlies" did not exist in **df1**, so this row received a value of **null** in the **points** column of the final joined DataFrame.

How to Perform an Anti-Join in PySpark

An **anti-join** allows you to return all rows in one DataFrame that do not have matching values in another DataFrame.

You can use the following syntax to perform an anti-join between two PySpark DataFrames:

```
df_anti_join = df1.join(df2, on=['team'], how='left_anti')
```

This particular example will perform an anti-join using the DataFrames named **df1** and **df2** and will only return the rows from **df1** where the value in the **team** column does not belong in the **team** column of **df2**.

The following example shows how to use this syntax in practice.

Example: How to Perform an Anti-Join in PySpark

Suppose we have the following DataFrame named **df1**:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data1 = [['A', 18],
       ['B', 22],
       ['C', 19],
       ['D', 14],
       ['E', 30]]

#define column names
columns1 = ['team', 'points']

#create dataframe using data and column names
df1 = spark.createDataFrame(data1, columns1)

#view dataframe
df1.show()

+----+------+
|team|points|
+----+------+
|   A|    18|
|   B|    22|
|   C|    19|
|   D|    14|
|   E|    30|
+----+------+
```

And suppose we have another DataFrame named **df2**:

```
#define data
data2 = [['A', 18],
       ['B', 22],
```

```
       ['C', 19],
       ['F', 22],
       ['G', 29]]

#define column names
columns2 = ['team', 'points']

#create dataframe using data and column names
df2 = spark.createDataFrame(data2, columns2)

#view dataframe
df2.show()

+----+------+
|team|points|
+----+------+
|   A|    18|
|   B|    22|
|   C|    19|
|   F|    22|
|   G|    29|
+----+------+
```

We can use the following syntax to perform an anti-join and return all rows in the first DataFrame that do not have a matching team in the second DataFrame:

```
#perform anti-join
df_anti_join = df1.join(df2, on=['team'], how='left_anti')

#view resulting DataFrame
df_anti_join.show()

+----+------+
|team|points|
+----+------+
|   D|    14|
|   E|    30|
+----+------+
```

We can see that there are exactly two teams from the first DataFrame that do not have a matching team name in the second DataFrame.

The anti-join worked as expected.

The end result is one DataFrame that only contains the rows where the team name belongs to the first DataFrame but not the second DataFrame.

How to Do an Outer Join in PySpark (With Example)

You can use the following basic syntax to perform an outer join in PySpark:

```
df_joined = df1.join(df2, on=['team'], how='full').show()
```

This particular example will perform an outer join using the DataFrames named **df1** and **df2** by joining on the column named **team**.

All rows from both DataFrames will be returned in the final DataFrame.

The following example shows how to use this syntax in practice.

Example: How to Do an Outer Join in PySpark

Suppose we have the following DataFrame named **df1**:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data1 = [['Mavs', 11],
       ['Hawks', 25],
       ['Nets', 32],
       ['Kings', 15],
       ['Warriors', 22],
       ['Suns', 17]]

#define column names
columns1 = ['team', 'points']

#create dataframe using data and column names
df1 = spark.createDataFrame(data1, columns1)

#view dataframe
df1.show()

+--------+------+
|    team|points|
+--------+------+
|    Mavs|    11|
|   Hawks|    25|
```

```
|    Nets|    32|
|   Kings|    15|
|Warriors|    22|
|    Suns|    17|
+--------+------+
```

And suppose we have another DataFrame named **df2**:

```
#define data
data2 = [['Mavs', 4],
       ['Nets', 7],
       ['Suns', 8],
       ['Grizzlies', 12],
       ['Kings', 7]]

#define column names
columns2 = ['team', 'assists']

#create dataframe using data and column names
df2 = spark.createDataFrame(data2, columns2)

#view dataframe
df2.show()

+---------+-------+
|     team|assists|
+---------+-------+
|     Mavs|      4|
|     Nets|      7|
|     Suns|      8|
|Grizzlies|     12|
|    Kings|      7|
+---------+-------+
```

We can use the following syntax to perform an outer join between these two DataFrames by joining on values from the **team** column:

```
#perform outer join using 'team' column
df_joined = df1.join(df2, on=['team'], how='full').show()

+---------+------+-------+
|     team|points|assists|
+---------+------+-------+
|Grizzlies|  null|     12|
|    Hawks|    25|   null|
|    Kings|    15|      7|
|     Mavs|    11|      4|
|     Nets|    32|      7|
|     Suns|    17|      8|
```

```
| Warriors|    22|   null|
+---------+------+-------+
```

Notice that the resulting DataFrame contains all rows from both DataFrames.

Note that if a team name didn't appear in both DataFrames, then a value of **null** appeared in either the **points** or **assists** column.

For example, the team name "Hawks" did not exist in **df2**, so this row received a value of **null** in the **assists** column of the final joined DataFrame.

How to Do an Inner Join in PySpark (With Example)

You can use the following basic syntax to perform an inner join in PySpark:

```
df_joined = df1.join(df2, on=['team'], how='inner').show()
```

This particular example will perform an inner join using the DataFrames named **df1** and **df2** by joining on the column named **team**.

The following example shows how to use this syntax in practice.

Example: How to Do an Inner Join in PySpark

Suppose we have the following DataFrame named **df1**:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data1 = [['Mavs', 11],
         ['Hawks', 25],
         ['Nets', 32],
         ['Kings', 15],
         ['Warriors', 22],
         ['Suns', 17]]

#define column names
columns1 = ['team', 'points']

#create dataframe using data and column names
df1 = spark.createDataFrame(data1, columns1)
```

```
#view dataframe
df1.show()

+--------+------+
|    team|points|
+--------+------+
|    Mavs|    11|
|   Hawks|    25|
|    Nets|    32|
|   Kings|    15|
|Warriors|    22|
|    Suns|    17|
+--------+------+
```

And suppose we have another DataFrame named **df2**:

```
#define data
data2 = [['Mavs', 4],
         ['Nets', 7],
         ['Suns', 8],
         ['Grizzlies', 12],
         ['Kings', 7]]

#define column names
columns2 = ['team', 'assists']

#create dataframe using data and column names
df2 = spark.createDataFrame(data2, columns2)

#view dataframe
df2.show()

+---------+-------+
|     team|assists|
+---------+-------+
|     Mavs|      4|
|     Nets|      7|
|     Suns|      8|
|Grizzlies|     12|
|    Kings|      7|
+---------+-------+
```

We can use the following syntax to perform an inner join between these two DataFrames by joining on values from the **team** column:

```
#perform inner join using 'team' column
df_joined = df1.join(df2, on=['team'], how='inner').show()
```

```
+-----+------+-------+
| team|points|assists|
+-----+------+-------+
|Kings|    15|      7|
| Mavs|    11|      4|
| Nets|    32|      7|
| Suns|    17|      8|
+-----+------+-------+
```

Notice that the resulting DataFrame only contains the rows where the **team** value occurred in both DataFrames.

For example, the team names **Kings**, **Mavs**, **Nets** and **Suns** all occurred in both DataFrames, which is why these were the four rows that were included in the final joined DataFrame.

PySpark: How to Join on Different Column Names

You can use the following syntax to join two DataFrames together based on different column names in PySpark:

```
df3 = df1.withColumn('id',
col('team_id')).join(df2.withColumn('id', col('team_name')),
on='id')
```

Here is what this syntax does:

- First, it renames the **team_id** column from **df1** to **id**.
- Then, it renames the **team_name** column from **df2** to **id**.
- Lastly, it joins together **df1** and **df2** based on values in the **id** columns.

The following example shows how to use this syntax in practice.

Example: How to Join on Different Column Names in PySpark

Suppose we have the following DataFrame named **df1**:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
```

```
data = [['Mavs', 18],
        ['Nets', 33],
        ['Lakers', 12],
        ['Kings', 15],
        ['Hawks', 19],
        ['Wizards', 24],
        ['Magic', 28]]

#define column names
columns = ['team_ID', 'points']

#create dataframe using data and column names
df1 = spark.createDataFrame(data, columns)

#view dataframe
df1.show()

+-------+------+
|team_ID|points|
+-------+------+
|   Mavs|    18|
|   Nets|    33|
| Lakers|    12|
|  Kings|    15|
|  Hawks|    19|
|Wizards|    24|
|  Magic|    28|
+-------+------+
```

And suppose we have another DataFrame named **df2**:

```
#define data
data = [['Hawks', 4],
        ['Wizards', 5],
        ['Raptors', 5],
        ['Kings', 12],
        ['Mavs', 7],
        ['Nets', 11],
        ['Magic', 3]]

#define column names
columns = ['team_name', 'assists']

#create dataframe using data and column names
df2 = spark.createDataFrame(data, columns)

#view dataframe
df2.show()
```

```
+---------+-------+
|team_name|assists|
+---------+-------+
|    Hawks|      4|
|  Wizards|      5|
|  Raptors|      5|
|    Kings|     12|
|     Mavs|      7|
|     Nets|     11|
|    Magic|      3|
+---------+-------+
```

We can use the following syntax to perform an inner join between these two DataFrames by renaming the team columns from each DataFrame to **id** and then by joining on values from the **id** column:

```
#join df1 and df2 on different column names
df3 = df1.withColumn('id',
col('team_id')).join(df2.withColumn('id', col('team_name')),
on='id')

#view resulting DataFrame
df3.show()

+-------+-------+------+---------+-------+
|     id|team_ID|points|team_name|assists|
+-------+-------+------+---------+-------+
|  Hawks|  Hawks|    19|    Hawks|      4|
|  Kings|  Kings|    15|    Kings|     12|
|  Magic|  Magic|    28|    Magic|      3|
|   Mavs|   Mavs|    18|     Mavs|      7|
|   Nets|   Nets|    33|     Nets|     11|
|Wizards|Wizards|    24|  Wizards|      5|
+-------+-------+------+---------+-------+
```

We have successfully joined the two DataFrames into one DataFrame based on matching values in the new **id** column.

Note that you can also use the **select** function to only display certain columns in the resulting joined DataFrame.

For example, we can use the following syntax to only display the **id**, **points** and **assists** columns in the joined DataFrame:

```
#join df1 and df2 on different column names
df3 = df1.withColumn('id',
col('team_id')).join(df2.withColumn('id', col('team_name')),
on='id')\
```

```
          .select('id', 'points', 'assists')

#view resulting DataFrame
df3.show()

+-------+------+-------+
|     id|points|assists|
+-------+------+-------+
|  Hawks|    19|      4|
|  Kings|    15|     12|
|  Magic|    28|      3|
|   Mavs|    18|      7|
|   Nets|    33|     11|
|Wizards|    24|      5|
+-------+------+-------+
```

Notice that only the **id**, **points** and **assists** columns are shown in the joined DataFrame.

How to Print One Column of a PySpark DataFrame

You can use the following methods to print one specific column of a PySpark DataFrame:

Method 1: Print Column Values with Column Name

```
df.select('my_column').show()
```

Method 2: Print Column Values Only

```
df.select('my_column').rdd.flatMap(list).collect()
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2]]
```

```
#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Print Column Values with Column Name

We can use the following syntax to print the column values along with the column name for the **conference** column of the DataFrame:

```
#print 'conference' column (with column name)
df.select('conference').show()

+----------+
|conference|
+----------+
|      East|
|      East|
|      East|
|      West|
|      West|
|      East|
+----------+
```

Notice that both the column name and the column values are printed for only the **conference** column of the DataFrame.

Example 2: Print Column Values Only

We can use the following syntax to print only the column values of the **conference** column of the DataFrame:

```
#print values only from 'conference' column
df.select('conference').rdd.flatMap(list).collect()

['East', 'East', 'East', 'West', 'West', 'East']
```

Notice that only the values from the **conference** column are printed and the name of the column is not included.

PySpark: How to Check if Column Contains String

You can use the following methods to check if a column of a PySpark DataFrame contains a string:

Method 1: Check if Exact String Exists in Column

```
#check if 'conference' column contains exact string 'Eas' in any
row
df.where(df.conference=='Eas').count()>0
```

Method 2: Check if Partial String Exists in Column

```
#check if 'conference' column contains partial string 'Eas' in any
row
df.filter(df.conference.contains('Eas')).count()>0
```

Method 3: Count Occurrences of Partial String in Column

```
#count occurrences of partial string 'Eas' in 'conference' column
df.filter(df.conference.contains('Eas')).count()
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11],
        ['A', 'East', 8],
        ['A', 'East', 10],
        ['B', 'West', 6],
        ['B', 'West', 6],
        ['C', 'East', 5]]
```

```
#define column names
columns = ['team', 'conference', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
|   A|      East|    11|
|   A|      East|     8|
|   A|      East|    10|
|   B|      West|     6|
|   B|      West|     6|
|   C|      East|     5|
+----+----------+------+
```

Example 1: Check if Exact String Exists in Column

The following code shows how to check if the exact string 'Eas' exists in the **conference** column of the DataFrame:

```
#check if 'conference' column contains exact string 'Eas' in any
row
df.where(df.conference=='Eas').count()>0

False
```

The output returns **False**, which tells us that the exact string 'Eas' does not exist in the **conference** column of the DataFrame.

Example 2: Check if Partial String Exists in Column

The following code shows how to check if the partial string 'Eas' exists in the **conference** column of the DataFrame:

```
#check if 'conference' column contains partial string 'Eas' in any
row
df.filter(df.conference.contains('Eas')).count()>0

True
```

The output returns **True**, which tells us that the partial string ‘Eas’ does exist in the **conference** column of the DataFrame.

Example 3: Count Occurrences of Partial String in Column

The following code shows how to count the number of times the partial string ‘Eas’ occurs in the **conference** column of the DataFrame:

```
#count occurrences of partial string 'Eas' in 'conference' column
df.filter(df.conference.contains('Eas')).count()

4
```

The output returns **4**, which tells us that the partial string ‘Eas’ occurs 4 times in the **conference** column of the DataFrame.

PySpark: How to Check if Column Exists in DataFrame

You can use the following methods in PySpark to check if a particular column exists in a DataFrame:

Method 1: Check if Column Exists (Case-Sensitive)

```
'points' in df.columns
```

Method 2: Check if Column Exists (Not Case-Sensitive)

```
'points'.upper() in (name.upper() for name in df.columns)
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', None, 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', None, 12],
        ['B', 'West', None, 4],
        ['C', 'East', 5, 2]]
```

```
#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      null|     8|      9|
|   A|      East|    10|      3|
|   B|      West|  null|     12|
|   B|      West|  null|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Check if Column Exists (Case-Sensitive)

We can use the following syntax to check if the column name **points** exists in the DataFrame:

```
#check if column name 'points' exists in the DataFrame
'points' in df.columns

True
```

The output returns **True** since the column name **points** does indeed exist in the DataFrame.

Note that this syntax is case-sensitive so if we search instead for the column name **Points** then we will receive an output of **False** since the case we searched for doesn't precisely match the case of the column name in the DataFrame:

```
#check if column name 'Points' exists in the DataFrame
'Points' in df.columns

False
```

Example 2: Check if Column Exists (Not Case-Sensitive)

We can use the following syntax to check if the column name **Points** exists in the DataFrame:

```
#check if column name 'Points' exists in the DataFrame
'Points'.upper() in (name.upper() for name in df.columns)
```

```
True
```

The output returns **True** even though the case of the column name that we searched for didn't precisely match the column name of **points** in the DataFrame.

Note: In this example we used the **upper()** function to first convert our search phrase to all uppercase and convert all column names in the DataFrame to uppercase.

This allowed us to perform a case-insensitive search.

PySpark: How to Check if DataFrame is Empty

You can use the following syntax to check if a PySpark DataFrame is empty:

```
print(df.count() == 0)
```

This will return **True** if the DataFrame is empty or **False** if the DataFrame is not empty.

Note that **df.count()** will count the number of rows in the DataFrame, so we're effectively checking if the total rows is equal to zero or not.

The following examples show how to use this syntax in practice.

Example 1: Check if Empty DataFrame is Empty

Suppose we create the following empty PySpark DataFrame with specific column names:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

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

#create empty RDD
empty_rdd=spark.sparkContext.emptyRDD()

#specify colum names and types
my_columns=[StructField('team', StringType(),True),
            StructField('position', StringType(),True),
            StructField('points', FloatType(),True)]
```

```
#create DataFrame with specific column names
df=spark.createDataFrame([], schema=StructType(my_columns))

#view DataFrame
df.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
+----+--------+------+
```

We can use the following syntax to check if the DataFrame is empty:

```
#check if DataFrame is empty
print(df.count() == 0)

True
```

We receive a value of **True**, which indicates that the DataFrame is indeed empty.

Example 2: Check if Non-Empty DataFrame is Empty

Suppose we create the following PySpark DataFrame that contains information about various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['Mavs', 18],
        ['Nets', 33],
        ['Lakers', 12],
        ['Mavs', 15],
        ['Cavs', 19],
        ['Wizards', 24],]

#define column names
columns = ['team', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+-------+------+
|   team|points|
+-------+------+
```

```
|   Mavs|    18|
|   Nets|    33|
| Lakers|    12|
|   Mavs|    15|
|   Cavs|    19|
|Wizards|    24|
+-------+------+
```

We can use the following syntax to check if the DataFrame is empty:

```
#check if DataFrame is empty
print(df.count() == 0)

False
```

We receive a value of **False**, which indicates that the DataFrame is not empty.

PySpark: How to Check Data Type of Columns in DataFrame

You can use the following methods in PySpark to check the data type of columns in a DataFrame:

Method 1: Check Data Type of One Specific Column

```
#return data type of 'conference' column
dict(df.dtypes)['conference']
```

Method 2: Check Data Type of All Columns

```
#return data type of all columns
df.dtypes
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', None, 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', None, 12],
```

```
        ['B', 'West', None, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      null|     8|      9|
|   A|      East|    10|      3|
|   B|      West|  null|     12|
|   B|      West|  null|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Check Data Type of One Specific Column

We can use the following syntax to check the data type of the **conference** column in the DataFrame:

```
#return data type of 'conference' column
dict(df.dtypes)['conference']

'string'
```

The output tells us that the **conference** column has a data type of **string**.

To check the data type of another specific column, simply replace **conference** with a different column name:

```
#return data type of 'points' column
dict(df.dtypes)['points']

'bigint'
```

The output tells us that the **points** column has a data type of **bigint**.

Example 2: Check Data Type of All Columns

We can use the following syntax to check the data type of all columns in the DataFrame:

```
#return data type of all columns
df.dtypes

[('team', 'string'),
 ('conference', 'string'),
 ('points', 'bigint'),
 ('assists', 'bigint')]
```

The output shows each of the column names along with the data type of each column.

For example, we can see:

- The **team** column has a data type of **string**.
- The **conference** column has a data type of **string**.
- The **points** column has a data type of **bigint**.
- The **assists** column has a data type of **bigint**.

And so on.

PySpark: How to Drop Multiple Columns from DataFrame

There are two common ways to drop multiple columns in a PySpark DataFrame:

Method 1: Drop Multiple Columns by Name

```
#drop 'team' and 'points' columns
df.drop('team', 'points').show()
```

Method 2: Drop Multiple Columns Based on List

```
#define list of columns to drop
drop_cols = ['team', 'points']

#drop all columns in list
df.select(*drop_cols).show()
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Drop Multiple Columns by Name

We can use the following syntax to drop the **team** and **points** columns from the DataFrame:

```
#drop 'team' and 'points' columns
df.drop('team', 'points').show()

+----------+-------+
|conference|assists|
+----------+-------+
|      East|      4|
|      East|      9|
|      East|      3|
|      West|     12|
|      West|      4|
|      East|      2|
+----------+-------+
```

Notice that the **team** and **points** columns have both been dropped from the DataFrame, just as we specified.

Example 2: Drop Multiple Columns Based on List

We can use the following syntax to specify a list of column names and then drop all columns in the DataFrame that belong to the list:

```
#define list of columns to drop
drop_cols = ['team', 'points']

#drop all columns in list
df.drop(*drop_cols).show()

+----------+-------+
|conference|assists|
+----------+-------+
|      East|      4|
|      East|      9|
|      East|      3|
|      West|     12|
|      West|      4|
|      East|      2|
+----------+-------+
```

Notice that the resulting DataFrame drops each of the column names that we specified in the list.

How to Drop Duplicate Rows from DataFrame

There are three common ways to drop duplicate rows from a PySpark DataFrame:

Method 1: Drop Rows with Duplicate Values Across All Columns

```
#drop rows that have duplicate values across all columns
df_new = df.dropDuplicates()
```

Method 2: Drop Rows with Duplicate Values Across Specific Columns

```
#drop rows that have duplicate values across 'team' and 'position'
columns
df_new = df.dropDuplicates(['team', 'position'])
```

Method 3: Drop Rows with Duplicate Values in One Specific Column

```
#drop rows that have duplicate values in 'team' column
df_new = df.dropDuplicates(['team'])
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'Guard', 11],
        ['A', 'Guard', 8],
        ['A', 'Forward', 22],
        ['A', 'Forward', 22],
        ['B', 'Guard', 14],
        ['B', 'Guard', 14],
        ['B', 'Forward', 13],
        ['B', 'Forward', 7]]

#define column names
columns = ['team', 'position', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
|   A|   Guard|    11|
|   A|   Guard|     8|
|   A| Forward|    22|
|   A| Forward|    22|
|   B|   Guard|    14|
|   B|   Guard|    14|
|   B| Forward|    13|
|   B| Forward|     7|
+----+--------+------+
```

Example 1: Drop Rows with Duplicate Values Across All Columns

We can use the following syntax to drop rows that have duplicate values across all columns in the DataFrame:

```
#drop rows that have duplicate values across all columns
df_new = df.dropDuplicates()

#view DataFrame without duplicates
df_new.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
|   A|   Guard|    11|
|   A|   Guard|     8|
|   A| Forward|    22|
|   B|   Guard|    14|
|   B| Forward|    13|
|   B| Forward|     7|
+----+--------+------+
```

A total of two rows were dropped from the DataFrame.

Example 2: Drop Rows with Duplicate Values Across Specific Columns

We can use the following syntax to drop rows that have duplicate values across the **team** and **position** columns in the DataFrame:

```
#drop rows that have duplicate values across 'team' and 'position'
columns
df_new = df.dropDuplicates(['team', 'position'])

#view DataFrame without duplicates
df_new.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
|   A|   Guard|    11|
|   A| Forward|    22|
|   B|   Guard|    14|
|   B| Forward|    13|
+----+--------+------+
```

Notice that the resulting DataFrame has no rows with duplicate values across both the **team** and **position** columns.

Example 3: Drop Rows with Duplicate Values in One Specific Column

We can use the following syntax to drop rows that have duplicate values in the **team** column of the DataFrame:

```
#drop rows that have duplicate values in 'team' column
df_new = df.dropDuplicates(['team'])

#view DataFrame without duplicates
df_new.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
|   A|   Guard|    11|
|   B|   Guard|    14|
+----+--------+------+
```

Notice that the resulting DataFrame has no rows with duplicate values in the **team** column.

Note: When duplicate rows are identified, only the first duplicate row is kept in the DataFrame while all other duplicate rows are dropped.

How to Select Distinct Rows in PySpark (With Examples)

You can use the following methods to select distinct rows in a PySpark DataFrame:

Method 1: Select Distinct Rows in DataFrame

```
#display distinct rows only
df.distinct().show()
```

Method 2: Select Distinct Values from Specific Column

```
#display distinct values from 'team' column only
df.select('team').distinct().show()
```

Method 3: Count Distinct Rows in DataFrame

```
#count number of distinct rows
```

```
df.distinct().count()
```

The following examples show how to use each of these methods in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'Guard', 11],
        ['A', 'Guard', 8],
        ['A', 'Forward', 22],
        ['A', 'Forward', 22],
        ['B', 'Guard', 14],
        ['B', 'Guard', 14],
        ['B', 'Forward', 13],
        ['B', 'Forward', 7]]

#define column names
columns = ['team', 'position', 'points']

#create DataFrame using data and column names
df = spark.createDataFrame(data, columns)

#view DataFrame
df.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
|   A|   Guard|    11|
|   A|   Guard|     8|
|   A| Forward|    22|
|   A| Forward|    22|
|   B|   Guard|    14|
|   B|   Guard|    14|
|   B| Forward|    13|
|   B| Forward|     7|
+----+--------+------+
```

Example 1: Select Distinct Rows in DataFrame

We can use the following syntax to select the distinct rows in the DataFrame:

```
#display distinct rows only
df.distinct().show()

+----+--------+------+
```

```
|team|position|points|
+----+--------+------+
|   A|   Guard|    11|
|   A|   Guard|     8|
|   A| Forward|    22|
|   B|   Guard|    14|
|   B| Forward|    13|
|   B| Forward|     7|
+----+--------+------+
```

Notice that each row in the resulting DataFrame is distinct.

Example 2: Select Distinct Values from Specific Column in DataFrame

We can use the following syntax to select the distinct values from the **team** column in the DataFrame:

```
#display distinct values from 'team' column only
df.select('team').distinct().show()

+----+
|team|
+----+
|   A|
|   B|
+----+
```

The output shows the two distinct values from the **team** column: **A** and **B**.

Example 3: Count Distinct Rows in DataFrame

We can use the following syntax to count the number of distinct rows in the DataFrame:

```
#count number of distinct rows
df.distinct().count()

6
```

The output tells us that there are **6** distinct rows in the entire DataFrame.

PySpark: How to Select Columns with Alias

There are two common ways to select columns and return aliased names in a PySpark DataFrame:

Method 1: Return One Column with Aliased Name

```
#select 'team' column and display using aliased name of 'team_name'
df.select(df.team.alias('team_name')).show()
```

Method 2: Return One Column with Aliased Name Along with All Other Columns

```
#select all columns and display 'team' column using aliased name of
'team_name'
df.withColumnRenamed('team', 'team_name').show()
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Return One Column with Aliased Name

We can use the following syntax to select the **team** column from the DataFrame and display it using the aliased name of **team_name**:

```
#select 'team' column and display using aliased name of 'team_name'
df.select(df.team.alias('team_name')).show()

+---------+
|team_name|
+---------+
|        A|
|        A|
|        A|
|        B|
|        B|
|        C|
+---------+
```

Notice that only the values from the **team** column are shown in the results and the column name is shown using the alias **team_name**.

Example 2: Return One Column with Aliased Name Along with All Other Columns

We can use the following syntax to select all columns from the DataFrame and display only the **team** column with an aliased name of **team_name**:

```
#select all columns and display 'team' column using aliased name of
'team_name'
df.withColumnRenamed('team', 'team_name').show()

+---------+----------+------+-------+
|team_name|conference|points|assists|
+---------+----------+------+-------+
|        A|      East|    11|      4|
|        A|      East|     8|      9|
|        A|      East|    10|      3|
|        B|      West|     6|     12|
|        B|      West|     6|      4|
|        C|      East|     5|      2|
+---------+----------+------+-------+
```

Notice that all columns from the DataFrame are returned and only the **team** column is displayed with an aliased name that we specified.

The function **withColumnRenamed** is particularly useful when you only want to display an aliased name for one column but you still want to include all other columns from the DataFrame in the output.

How to Select Top N Rows in PySpark DataFrame (With Examples)

There are two common ways to select the top N rows in a PySpark DataFrame:

Method 1: Use take()

```
df.take(10)
```

This method will return an **array** of the top 10 rows.

Method 2: Use limit()

```
df.limit(10).show()
```

This method will return a new **DataFrame** that contains the top 10 rows.

The following examples show how to use each of these methods in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11],
        ['A', 'East', 8],
        ['A', 'East', 10],
        ['B', 'West', 6],
        ['B', 'West', 6],
        ['C', 'East', 5]]

#define column names
columns = ['team', 'conference', 'points']

#create DataFrame using data and column names
df = spark.createDataFrame(data, columns)

#view DataFrame
df.show()
```

```
+----+----------+------+
|team|conference|points|
+----+----------+------+
|   A|      East|    11|
|   A|      East|     8|
|   A|      East|    10|
|   B|      West|     6|
|   B|      West|     6|
|   C|      East|     5|
+----+----------+------+
```

Example 1: Select Top N Rows Using take()

We can use the following syntax with the **take()** method to select the top 3 rows from the DataFrame:

```
#select top 3 rows from DataFrame
df.take(3)

[Row(team='A', conference='East', points=11, assists=4),
 Row(team='A', conference='East', points=8, assists=9),
 Row(team='A', conference='East', points=10, assists=3)]
```

This method returns an array of the top 3 rows of the DataFrame.

Example 2: Select Top N Rows Using limit()

We can use the following syntax with the **limit()** method to select the top 3 rows from the DataFrame:

```
#select top 3 rows from DataFrame
df.limit(3).show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
+----+----------+------+-------+
```

This method returns a DataFrame that contains only the top 3 rows of the original DataFrame.

Note that if you'd like to only select the top 3 rows for particular columns, you can specify those columns by using the **select()** function:

```
#select top 3 rows from DataFrame only for 'team' and 'points'
columns
df.select('team', 'points').limit(3).show()

+----+------+
|team|points|
+----+------+
|   A|    11|
|   A|     8|
|   A|    10|
+----+------+
```

Notice that only the top 3 rows for the **team** and **points** columns are shown in the resulting DataFrame.

PySpark: Select All Columns Except Specific Ones

The easiest way to select all columns except specific ones in a PySpark DataFrame is by using the **drop** function.

Here are two common ways to do so:

Method 1: Select All Columns Except One

```
#select all columns except 'conference' column
df.drop('conference').show()
```

Method 2: Select All Columns Except Several Specific Ones

```
#select all columns except 'conference' and 'assists' columns
df.drop('conference', 'assists').show()
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
```

```
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Select All Columns Except One

We can use the following syntax to select all columns in the DataFrame *except* for the **conference** column:

```
#select all columns except 'conference' column
df.drop('conference').show()

+----+------+-------+
|team|points|assists|
+----+------+-------+
|   A|    11|      4|
|   A|     8|      9|
|   A|    10|      3|
|   B|     6|     12|
|   B|     6|      4|
|   C|     5|      2|
+----+------+-------+
```

Notice that the resulting DataFrame contains all columns from the original DataFrame *except* for the **conference** column.

Example 2: Select All Columns Except Several Specific Ones

We can use the following syntax to select all columns in the DataFrame *except* for the **conference** and **assists** columns:

```
#select all columns except 'conference' and 'assists' column
df.drop('conference', 'assists').show()

+----+------+
|team|points|
+----+------+
|   A|    11|
|   A|     8|
|   A|    10|
|   B|     6|
|   B|     6|
|   C|     5|
+----+------+
```

Notice that the resulting DataFrame contains all columns from the original DataFrame *except* for the **conference** and **assists** columns.

How to Use a Case Statement in PySpark (With Example)

A **case statement** is a type of statement that goes through conditions and returns a value when the first condition is met.

The easiest way to implement a case statement in a PySpark DataFrame is by using the following syntax:

```
from pyspark.sql.functions import when

df.withColumn('class',when(df.points<9, 'Bad').when(df.points<12,
'OK').when(df.points<15, 'Good').otherwise('Great')).show()
```

This particular example adds a new column to a DataFrame called **class** that takes on the following values:

- **Bad** if the value in the points column is less than 9
- **OK** if the value in the points column is less than 12
- **Good** if the value in the points column is less than 15
- **Great** if none of the previous conditions are true

The following example shows how to use this function in practice.

Example: How to Use a Case Statement in PySpark

Suppose we have the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 6],
        ['B', 8],
        ['C', 9],
        ['D', 9],
        ['E', 12],
        ['F', 14],
        ['G', 15],
        ['H', 17],
        ['I', 19],
        ['J', 22]]

#define column names
columns = ['player', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+------+------+
|player|points|
+------+------+
|     A|     6|
|     B|     8|
|     C|     9|
|     D|     9|
|     E|    12|
|     F|    14|
|     G|    15|
|     H|    17|
|     I|    19|
|     J|    22|
+------+------+
```

We can use the following syntax to write a case statement that creates a new column called **class** whose values are determined by the values in the **points** column:

```
from pyspark.sql.functions import when
```

```
df.withColumn('class',when(df.points<9, 'Bad').when(df.points<12,
'OK').when(df.points<15, 'Good').otherwise('Great')).show()

+------+------+-----+
|player|points|class|
+------+------+-----+
|     A|     6|  Bad|
|     B|     8|  Bad|
|     C|     9|   OK|
|     D|     9|   OK|
|     E|    12| Good|
|     F|    14| Good|
|     G|    15|Great|
|     H|    17|Great|
|     I|    19|Great|
|     J|    22|Great|
+------+------+-----+
```

The case statement looked at the value in the **points** column and returned:

- **Bad** if the value in the points column was less than 9
- **OK** if the value in the points column was less than 12
- **Good** if the value in the points column was less than 15
- **Great** if none of the previous conditions are true

Note: We chose to use three conditions in this particular example but you can chain together as many **when()** statements as you'd like to include even more conditions in your own case statement.

PySpark: How to Convert Column from Date to String

You can use the following syntax to convert a column from a date to a string in PySpark:

```
from pyspark.sql.functions import date_format

df_new = df.withColumn('date_string', date_format('date',
'MM/dd/yyyy'))
```

This particular example converts the dates in the **date** column to strings in a new column called **date_string**, using **MM/dd/yyyy** as the date format.

The following example shows how to use this syntax in practice.

Example: How to Convert Column from Date to String in PySpark

Suppose we have the following PySpark DataFrame that contains information about sales made on various days for some company:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

import datetime

#define data
data = [[datetime.date(2023, 10, 30), 136],
        [datetime.date(2023, 11, 14), 223],
        [datetime.date(2023, 11, 22), 450],
        [datetime.date(2023, 11, 25), 290],
        [datetime.date(2023, 12, 19), 189]]

#define column names
columns = ['date', 'sales']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe with full column content
df.show()

+----------+-----+
|      date|sales|
+----------+-----+
|2023-10-30|  136|
|2023-11-14|  223|
|2023-11-22|  450|
|2023-11-25|  290|
|2023-12-19|  189|
+----------+-----+
```

We can use the **dtypes** function to check the data type of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('date', 'date'), ('sales', 'bigint')]
```

We can see that the **date** column currently has a data type of **date**.

To convert this column from a date to a string, we can use the following syntax:

```
from pyspark.sql.functions import date_format

#create new column that converts dates to strings
df_new = df.withColumn('date_string', date_format('date',
'MM/dd/yyyy'))

#view new DataFrame
df_new.show()

+----------+-----+-----------+
|      date|sales|date_string|
+----------+-----+-----------+
|2023-10-30|  136| 10/30/2023|
|2023-11-14|  223| 11/14/2023|
|2023-11-22|  450| 11/22/2023|
|2023-11-25|  290| 11/25/2023|
|2023-12-19|  189| 12/19/2023|
+----------+-----+-----------+
```

We can use the **dtypes** function once again to view the data types of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('date', 'date'), ('sales', 'bigint'), ('date_string', 'string')]
```

We can see that the **date_string** column has a data type of **string**.

We have successfully created a string column from a date column.

Note: We used **MM/dd/yyyy** as the date format within the **date_format** function but feel free to use whatever date format you'd like.

How to Convert String to Date in PySpark (With Example)

You can use the following syntax to convert a string column to a date column in a PySpark DataFrame:

```
from pyspark.sql import functions as F

df = df.withColumn('my_date_column', F.to_date('my_date_column'))
```

This particular example converts the values in the **my_date_column** from strings to dates.

The following example shows how to use this syntax in practice.

Example: How to Convert String to Date in PySpark

Suppose we have the following PySpark DataFrame that contains information about sales made on various dates at some company:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['2023-01-15', 225],
        ['2023-02-24', 260],
        ['2023-07-14', 413],
        ['2023-10-30', 368]]

#define column names
columns = ['date', 'sales']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----------+-----+
|      date|sales|
+----------+-----+
|2023-01-15|  225|
|2023-02-24|  260|
|2023-07-14|  413|
|2023-10-30|  368|
+----------+-----+
```

We can use the following syntax to display the data type of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('date', 'string'), ('sales', 'bigint')]
```

We can see that the date column currently has a data type of **string**.

To convert this column from a string to a date, we can use the following syntax:

```
from pyspark.sql import functions as F
```

```
#convert 'date' column from string to date
df = df.withColumn('date', F.to_date('date'))

#view updated DataFrame
df.show()

+----------+-----+
|      date|sales|
+----------+-----+
|2023-01-15|  225|
|2023-02-24|  260|
|2023-07-14|  413|
|2023-10-30|  368|
+----------+-----+
```

We can use the **dtypes** function once again to view the data types of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('date', 'date'), ('sales', 'bigint')]
```

We can see that the date column now has a data type of **date**.

We have successfully converted a string column to a date column.

How to Convert String to Timestamp in PySpark (With Example)

You can use the following syntax to convert a string column to a timestamp column in a PySpark DataFrame:

```
from pyspark.sql import functions as F

df = df.withColumn('ts_new', F.to_timestamp('ts', 'yyyy-MM-dd
HH:mm:ss'))
```

This particular example creates a new column called **ts_new** that contains timestamp values from the string values in the **ts** column.

The following example shows how to use this syntax in practice.

Example: How to Convert String to Timestamp in PySpark

Suppose we have the following PySpark DataFrame that contains information about sales made on various timestamps at some company:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['2023-01-15 04:14:22', 225],
        ['2023-02-24 10:55:01', 260],
        ['2023-07-14 18:34:59', 413],
        ['2023-10-30 22:20:05', 368]]

#define column names
columns = ['ts', 'sales']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+-------------------+-----+
|                 ts|sales|
+-------------------+-----+
|2023-01-15 04:14:22|  225|
|2023-02-24 10:55:01|  260|
|2023-07-14 18:34:59|  413|
|2023-10-30 22:20:05|  368|
+-------------------+-----+
```

We can use the following syntax to display the data type of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('ts', 'string'), ('sales', 'bigint')]
```

We can see that the **ts** column currently has a data type of **string**.

To convert this column from a string to a timestamp, we can use the following syntax:

```
from pyspark.sql import functions as F

#convert 'ts' column from string to timestamp
df = df.withColumn('ts_new', F.to_timestamp('ts', 'yyyy-MM-dd
HH:mm:ss'))
```

```
#view updated DataFrame
df.show()

+-------------------+-----+-------------------+
|                 ts|sales|             ts_new|
+-------------------+-----+-------------------+
|2023-01-15 04:14:22|  225|2023-01-15 04:14:22|
|2023-02-24 10:55:01|  260|2023-02-24 10:55:01|
|2023-07-14 18:34:59|  413|2023-07-14 18:34:59|
|2023-10-30 22:20:05|  368|2023-10-30 22:20:05|
+-------------------+-----+-------------------+
```

We can use the **dtypes** function once again to view the data types of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('ts', 'string'), ('sales', 'bigint'), ('ts_new', 'timestamp')]
```

We can see that the new column called **ts_new** has a data type of **timestamp**.

We have successfully converted a string column to a timestamp column.

How to Convert Timestamp to Date in PySpark (With Example)

You can use the following syntax to convert a timestamp column to a date column in a PySpark DataFrame:

```
from pyspark.sql.types import DateType

df = df.withColumn('my_date', df['my_timestamp'].cast(DateType()))
```

This particular example creates a new column called **my_date** that contains the date values from the timestamp values in the **my_timestamp** column.

The following example shows how to use this syntax in practice.

Example: How to Convert Timestamp to Date in PySpark

Suppose we have the following PySpark DataFrame that contains information about sales made on various timestamps at some company:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

from pyspark.sql import functions as F

#define data
data = [['2023-01-15 04:14:22', 225],
        ['2023-02-24 10:55:01', 260],
        ['2023-07-14 18:34:59', 413],
        ['2023-10-30 22:20:05', 368]]

#define column names
columns = ['ts', 'sales']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#convert string column to timestamp
df = df.withColumn('ts', F.to_timestamp('ts', 'yyyy-MM-dd
HH:mm:ss'))

#view dataframe
df.show()

+-------------------+-----+
|                 ts|sales|
+-------------------+-----+
|2023-01-15 04:14:22|  225|
|2023-02-24 10:55:01|  260|
|2023-07-14 18:34:59|  413|
|2023-10-30 22:20:05|  368|
+-------------------+-----+
```

We can use the following syntax to display the data type of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('ts', 'timestamp'), ('sales', 'bigint')]
```

We can see that the **ts** column currently has a data type of **timestamp**.

To convert this column from a timestamp to a date, we can use the following syntax:

```
from pyspark.sql.types import DateType
```

```
#create date column from timestamp column
df = df.withColumn('new_date', df['ts'].cast(DateType()))

#view updated DataFrame
df.show()

+-------------------+-----+----------+
|                 ts|sales|  new_date|
+-------------------+-----+----------+
|2023-01-15 04:14:22|  225|2023-01-15|
|2023-02-24 10:55:01|  260|2023-02-24|
|2023-07-14 18:34:59|  413|2023-07-14|
|2023-10-30 22:20:05|  368|2023-10-30|
+-------------------+-----+----------+
```

We can use the **dtypes** function once again to view the data types of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('ts', 'timestamp'), ('sales', 'bigint'), ('new_date', 'date')]
```

We can see that the **new_date** column has a data type of **date**.

We have successfully created a date column from a timestamp column.

How to Convert String to Integer in PySpark (With Example)

You can use the following syntax to convert a string column to an integer column in a PySpark DataFrame:

```
from pyspark.sql.types import IntegerType

df = df.withColumn('my_integer',
df['my_string'].cast(IntegerType()))
```

This particular example creates a new column called **my_integer** that contains the integer values from the string values in the **my_string** column.

The following example shows how to use this syntax in practice.

Example: How to Convert String to Integer in PySpark

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', '11'],
        ['B', '19'],
        ['C', '22'],
        ['D', '25'],
        ['E', '12'],
        ['F', '41'],
        ['G', '32'],
        ['H', '20']]

#define column names
columns = ['team', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+------+
|team|points|
+----+------+
|   A|    11|
|   B|    19|
|   C|    22|
|   D|    25|
|   E|    12|
|   F|    41|
|   G|    32|
|   H|    20|
+----+------+
```

We can use the following syntax to display the data type of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('team', 'string'), ('points', 'string')]
```

We can see that the **points** column currently has a data type of **string**.

To convert this column from a string to an integer, we can use the following syntax:

```
from pyspark.sql.types import IntegerType

#create integer column from string column
df = df.withColumn('points_integer',
df['points'].cast(IntegerType()))

#view updated DataFrame
df.show()

+----+------+--------------+
|team|points|points_integer|
+----+------+--------------+
|   A|    11|            11|
|   B|    19|            19|
|   C|    22|            22|
|   D|    25|            25|
|   E|    12|            12|
|   F|    41|            41|
|   G|    32|            32|
|   H|    20|            20|
+----+------+--------------+
```

We can use the **dtypes** function once again to view the data types of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('team', 'string'), ('points', 'string'), ('points_integer',
'int')]
```

We can see that the **points_integer** column has a data type of **int**.

We have successfully created an integer column from a string column.

How to Convert Integer to String in PySpark (With Example)

You can use the following syntax to convert an integer column to a string column in a PySpark DataFrame:

```
from pyspark.sql.types import StringType
```

```
df = df.withColumn('my_string',
df['my_integer'].cast(StringType()))
```

This particular example creates a new column called **my_string** that contains the string values from the integer values in the **my_integer** column.

The following example shows how to use this syntax in practice.

Example: How to Convert Integer to String in PySpark

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 11],
        ['B', 19],
        ['C', 22],
        ['D', 25],
        ['E', 12],
        ['F', 41],
        ['G', 32],
        ['H', 20]]

#define column names
columns = ['team', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+------+
|team|points|
+----+------+
|   A|    11|
|   B|    19|
|   C|    22|
|   D|    25|
|   E|    12|
|   F|    41|
|   G|    32|
|   H|    20|
+----+------+
```

We can use the following syntax to display the data type of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('team', 'string'), ('points', 'bigint')]
```

We can see that the **points** column currently has a data type of **integer**.

To convert this column from an integer to a string, we can use the following syntax:

```
from pyspark.sql.types import StringType

#create string column from integer column
df = df.withColumn('points_string',
df['points'].cast(StringType()))

#view updated DataFrame
df.show()

+----+------+-------------+
|team|points|points_string|
+----+------+-------------+
|   A|    11|           11|
|   B|    19|           19|
|   C|    22|           22|
|   D|    25|           25|
|   E|    12|           12|
|   F|    41|           41|
|   G|    32|           32|
|   H|    20|           20|
+----+------+-------------+
```

We can use the **dtypes** function once again to view the data types of each column in the DataFrame:

```
#check data type of each column
df.dtypes

[('team', 'string'), ('points', 'bigint'), ('points_string',
'string')]
```

We can see that the **points_string** column has a data type of **string**.

We have successfully created a string column from an integer column.

PySpark: How to Convert RDD to DataFrame (With Example)

You can use the **toDF()** function to convert a RDD (resilient distributed dataset) to a DataFrame in PySpark:

```
my_df = my_RDD.toDF()
```

This particular example will convert the RDD named **my_RDD** to a DataFrame called **my_df**.

The following example shows how to use this syntax in practice.

Example: How to Convert RDD to DataFrame in PySpark

First, let's create the following RDD:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [('A', 11),
        ('B', 19),
        ('C', 22),
        ('D', 25),
        ('E', 12),
        ('F', 41)]

#create RDD using data
my_RDD = spark.sparkContext.parallelize(data)
```

We can verify that this object is a RDD by using the **type()** function:

```
#check object type
type(my_RDD)

pyspark.rdd.RDD
```

We can see that the object **my_RDD** is indeed a RDD.

We can then use the following syntax to convert the RDD to a PySpark DataFrame:

```
#convert RDD to DataFrame
my_df = my_RDD.toDF()
```

```
#view DataFrame
my_df.show()

+---+---+
| _1| _2|
+---+---+
|  A| 11|
|  B| 19|
|  C| 22|
|  D| 25|
|  E| 12|
|  F| 41|
+---+---+
```

We can see that the RDD has been converted to a DataFrame.

We can verify that the **my_df** object is a DataFrame by using the **type()** function once again:

```
#check object type
type(my_df)

pyspark.sql.dataframe.DataFrame
```

We can see that the object **my_df** is indeed a DataFrame.

Note that the **toDF()** function uses column names **_1** and **_2** by default.

However, we can also specify column names to use within the **toDF()** function:

```
#convert RDD to DataFrame with specific column names
my_df = my_RDD.toDF(['player', 'assists'])

#view DataFrame
my_df.show()

+------+-------+
|player|assists|
+------+-------+
|     A|     11|
|     B|     19|
|     C|     22|
|     D|     25|
|     E|     12|
|     F|     41|
+------+-------+
```

Notice that the RDD has now been converted to a DataFrame with the column names **player** and **assists**.

PySpark: How to Convert Column to Lowercase

You can use the following syntax to convert a column to lowercase in a PySpark DataFrame:

```
from pyspark.sql.functions import lower

df = df.withColumn('my_column', lower(df['my_column']))
```

The following example shows how to use this syntax in practice.

Example: How to Convert Column to Lowercase in PySpark

Suppose we create the following PySpark DataFrame that contains information about various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
```

```
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Suppose we would like to convert all strings in the **conference** column to lowercase.

We can use the following syntax to do so:

```
from pyspark.sql.functions import lower

#convert 'conference' column to lowercase
df = df.withColumn('conference', lower(df['conference']))

#view updated DataFrame
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      east|    11|      4|
|   A|      east|     8|      9|
|   A|      east|    10|      3|
|   B|      west|     6|     12|
|   B|      west|     6|      4|
|   C|      east|     5|      2|
+----+----------+------+-------+
```

Notice that all strings in the **conference** column of the updated DataFrame are now lowercase.

Note #1: We used the **withcolumn** function to return a new DataFrame with the **conference** column modified and all other columns left the same.

PySpark: How to Convert Column to Uppercase

You can use the following syntax to convert a column to uppercase in a PySpark DataFrame:

```
from pyspark.sql.functions import upper
```

```
df = df.withColumn('my_column', upper(df['my_column']))
```

The following example shows how to use this syntax in practice.

Example: How to Convert Column to Uppercase in PySpark

Suppose we create the following PySpark DataFrame that contains information about various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Suppose we would like to convert all strings in the **conference** column to uppercase.

We can use the following syntax to do so:

```
from pyspark.sql.functions import upper

#convert 'conference' column to uppercase
df = df.withColumn('conference', upper(df['conference']))
```

```
#view updated DataFrame
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      EAST|    11|      4|
|   A|      EAST|     8|      9|
|   A|      EAST|    10|      3|
|   B|      WEST|     6|     12|
|   B|      WEST|     6|      4|
|   C|      EAST|     5|      2|
+----+----------+------+-------+
```

Notice that all strings in the **conference** column of the updated DataFrame are now uppercase.

Note #1: We used the **withColumn** function to return a new DataFrame with the **conference** column modified and all other columns left the same.

How to Use "Is Not Null" in PySpark (With Examples)

You can use the following methods in PySpark to filter DataFrame rows where a value in a particular column is not null:

Method 1: Filter for Rows where Value is Not Null in Specific Column

```
#filter for rows where value is not null in 'points' column
df.filter(df.points.isNotNull()).show()
```

Method 2: Filter for Rows where Value is Not Null in Any Column

```
#filter for rows where value is not null in any column
df.dropna().show()
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
```

```
        ['A', None, 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', None, 12],
        ['B', 'West', None, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      null|     8|      9|
|   A|      East|    10|      3|
|   B|      West|  null|     12|
|   B|      West|  null|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Filter for Rows where Value is Not Null in Specific Column

We can use the following syntax to filter the DataFrame to only show rows where the value in the **points** column is not null:

```
#filter for rows where value is not null in 'points' column
df.filter(df.points.isNotNull()).show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      null|     8|      9|
|   A|      East|    10|      3|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

The resulting DataFrame only contains rows where the value in the **points** column is not null.

Example 2: Filter for Rows where Value is Not Null in Any Column

We can use the following syntax to filter the DataFrame to only show rows where there are no null values in any column:

```
#filter for rows where value is not null in any column
df.dropna().show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|    10|      3|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

The resulting DataFrame only contains rows where there are no null values in any column.

How to Use "IS NOT IN" in PySpark (With Example)

You can use the following syntax in PySpark to filter DataFrame rows where a value in a particular column is not in a particular list:

```
#define array of values
my_array = ['A', 'D', 'E']

#filter DataFrame to only contain rows where 'team' is not in
my_array
df.filter(~df.team.isin(my_array)).show()
```

This particular example will filter the DataFrame to only contain rows where the value in the **team** column is not equal to **A**, **D**, or **E**.

The following example shows how to use this syntax in practice.

Example: How to Use "IS NOT IN" in PySpark

Suppose we have the following PySpark DataFrame that contains information about various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2],
        ['D', 'East', 14, 2],
        ['E', 'West', 25, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
|   D|      East|    14|      2|
|   E|      West|    25|      2|
+----+----------+------+-------+
```

We can use the following syntax to filter the DataFrame to only show rows where the value in the **team** column is not equal to **A**, **D**, or **E**:

```
#define array of values
my_array = ['A', 'D', 'E']

#filter DataFrame to only contain rows where 'team' is not in
my_array
df.filter(~df.team.isin(my_array)).show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   B|      West|     6|     12|
```

```
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

The resulting DataFrame only contains rows where the value in the **team** column is not equal to **A**, **D**, or **E**.

Note: The tilde (~) operator is used in PySpark to represent **NOT**.

By using this operator along with the **isin** function, we are able to filter the DataFrame to only contain rows where the value in a particular column is not in a list of values.

How to Use "OR" Operator in PySpark (With Examples)

There are two common ways to filter a PySpark DataFrame by using an "OR" operator:

Method 1: Use "OR"

```
#filter DataFrame where points is greater than 9 or team equals "B"
df.filter('points>9 or team=="B"').show()
```

Method 2: Use | Symbol

```
#filter DataFrame where points is greater than 9 or team equals "B"
df.filter((df.points>9) | (df.team=="B")).show()
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']
```

```
#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Filter DataFrame Using "OR"

We can use the following syntax with the **filter** function and the word **or** to filter the DataFrame to only contain rows where the value in the **points** column is greater than 9 or the value in the **team** column is equal to B:

```
#filter DataFrame where points is greater than 9 or team equals "B"
df.filter('points>9 or team=="B"').show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
+----+----------+------+-------+
```

Notice that each of the rows in the resulting DataFrame meet at least one of the following conditions:

- The value in the points column is greater than 9
- The value in the team column is equal to "B"

Also note that in this example we only used one **or** operator but you can combine as many **or** operators as you'd like inside the **filter** function to filter using even more conditions.

Example 2: Filter DataFrame Using | Symbol

We can use the following syntax with the **filter** function and the | symbol to filter the DataFrame to only contain rows where the value in the **points** column is greater than 9 or the value in the **team** column is equal to B:

```
#filter DataFrame where points is greater than 9 or team equals "B"
df.filter((df.points>9) | (df.team=="B")).show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
+----+----------+------+-------+
```

Notice that each of the rows in the resulting DataFrame meet at least one of the following conditions:

- The value in the points column is greater than 9
- The value in the team column is equal to “B”

Also note that this DataFrame matches the DataFrame from the previous example.

How to Use “AND” Operator in PySpark (With Examples)

There are two common ways to filter a PySpark DataFrame by using an “AND” operator:

Method 1: Use “AND”

```
#filter DataFrame where points is greater than 5 and conference
equals "East"
df.filter('points>5 and conference=="East"').show()
```

Method 2: Use & Symbol

```
#filter DataFrame where points is greater than 5 and conference
equals "East"
df.filter((df.points>5) & (df.conference=="East")).show()
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Filter DataFrame Using "AND"

We can use the following syntax with the **filter** function and the word **and** to filter the DataFrame to only contain rows where the value in the **points** column is greater than 5 and the value in the **conference** column is equal to East:

```
#filter DataFrame where points is greater than 5 and conference
equals "East"
df.filter('points>5 and conference=="East"').show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
+----+----------+------+-------+
```

Notice that each of the rows in the resulting DataFrame meet both of the following conditions:

- The value in the points column is greater than 5
- The value in the conference column is equal to “East”

Also note that in this example we only used one **and** operator but you can combine as many **and** operators as you’d like inside the **filter** function to filter using even more conditions.

Example 2: Filter DataFrame Using & Symbol

We can use the following syntax with the **filter** function and the **&** symbol to filter the DataFrame to only contain rows where the value in the **points** column is greater than 5 and the value in the **conference** column is equal to East:

```
#filter DataFrame where points is greater than 5 and conference
equals "East"
df.filter((df.points>5) & (df.conference=="East")).show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
+----+----------+------+-------+
```

Notice that each of the rows in the resulting DataFrame meet both of the following conditions:

- The value in the points column is greater than 5
- The value in the conference column is equal to “East”

Also note that this DataFrame matches the DataFrame from the previous example.

How to Use “Not Equal” Operator in PySpark (With Examples)

There are two common ways to filter a PySpark DataFrame by using a “Not Equal” operator:

Method 1: Filter Using One “Not Equal” Operator

```
#filter DataFrame where team is not equal to 'A'
df.filter(df.team!='A').show()
```

Method 2: Filter Using Multiple "Not Equal" Operators

```
#filter DataFrame where team is not equal to 'A' and points is not
equal to 5
df.filter((df.team!='A') & (df.points!=5)).show()
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', 10, 3],
        ['B', 'West', 6, 12],
        ['B', 'West', 6, 4],
        ['C', 'East', 5, 2]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|    10|      3|
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Filter Using One "Not Equal" Operator

We can use the following syntax to filter the DataFrame to only contain rows where the **team** column is not equal to A:

```
#filter DataFrame where team is not equal to 'A'
df.filter(df.team!='A').show()
```

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   B|      West|     6|     12|
|   B|      West|     6|      4|
|   C|      East|     5|      2|
+----+----------+------+-------+
```

Notice that each of the rows in the resulting DataFrame contain a value in the **team** column that is not equal to A.

Example 2: Filter Using Multiple "Not Equal" Operators

We can use the following syntax to filter the DataFrame to only contain rows where the **team** column is not equal to A *and* the value in the **points** column is not equal to 5:

```
#filter DataFrame where team is not equal to 'A' and points is not
equal to 5
df.filter((df.team!='A') & (df.points!=5)).show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   B|      West|     6|     12|
|   B|      West|     6|      4|
+----+----------+------+-------+
```

Notice that each of the rows in the resulting DataFrame contain a value in the **team** column that is not equal to A *and* a value in the **points** column that is not equal to 5.

PySpark: How to Use Case-Insensitive "Contains"

By default, the **contains** function in PySpark is case-sensitive.

However, you can use the following syntax to use a case-insensitive "contains" to filter a DataFrame where rows contain a specific string, regardless of case:

```
from pyspark.sql.functions import upper

#perform case-insensitive filter for rows that contain 'AVS' in
team column
df.filter(upper(df.team).contains('AVS')).show()
```

The following example shows how to use this syntax in practice.

Example: How to Use Case-Insensitive "Contains" in PySpark

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['Mavs', 14],
        ['Nets', 22],
        ['Nets', 31],
        ['Cavs', 27],
        ['CAVS', 26],
        ['Spurs', 40],
        ['mavs', 23],
        ['MAVS', 17],]

#define column names
columns = ['team', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+-----+------+
| team|points|
+-----+------+
| Mavs|    14|
| Nets|    22|
| Nets|    31|
| Cavs|    27|
| CAVS|    26|
|Spurs|    40|
| mavs|    23|
| MAVS|    17|
+-----+------+
```

Suppose we use the following syntax to filter the DataFrame to only contain rows where the **team** column contains "AVS" somewhere in the string:

```
#filter DataFrame where team column contains 'AVS'
```

```
df.filter(df.team.contains('AVS')).show()

+----+------+
|team|points|
+----+------+
|CAVS|    26|
|MAVS|    17|
+----+------+
```

Notice that this syntax performs a **case-sensitive** search by default and only returns the rows where the **team** column contains "AVS" in all uppercase.

However, suppose we would like to perform a case-insensitive search and return all rows where the **team** column contains "AVS", regardless of case.

We can use the following syntax to do so:

```
from pyspark.sql.functions import upper

#perform case-insensitive filter for rows that contain 'AVS' in
team column
df.filter(upper(df.team).contains('AVS')).show()

+----+------+
|team|points|
+----+------+
|Mavs|    14|
|Cavs|    27|
|CAVS|    26|
|mavs|    23|
|MAVS|    17|
+----+------+
```

Notice that this syntax performs a **case-insensitive** search and returns all rows where the **team** column contains "AVS", regardless of case.

Note: We used the **upper** function to first convert all strings in the **team** column to uppercase and then searched for "AVS", which is the equivalent of using a case-sensitive "contains" filter.

PySpark: How to Filter for "Not Contains"

You can use the following syntax to filter a PySpark DataFrame by using a "Not Contains" operator:

```
#filter DataFrame where team does not contain 'avs'
```

```
df.filter(~df.team.contains('avs')).show()
```

The following example shows how to use this syntax in practice.

Example: How to Filter for “Not Contains” in PySpark

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['Mavs', 14],
        ['Nets', 22],
        ['Nets', 31],
        ['Cavs', 27],
        ['Kings', 26],
        ['Spurs', 40],
        ['Lakers', 23],
        ['Spurs', 17],]

#define column names
columns = ['team', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+------+------+
|  team|points|
+------+------+
|  Mavs|    14|
|  Nets|    22|
|  Nets|    31|
|  Cavs|    27|
| Kings|    26|
| Spurs|    40|
|Lakers|    23|
| Spurs|    17|
+------+------+
```

We can use the following syntax to filter the DataFrame to only contain rows where the **team** column does not contain “avs” anywhere in the string:

```
#filter DataFrame where team does not contain 'avs'
```

```
df.filter(~df.team.contains('avs')).show()

+------+------+
|  team|points|
+------+------+
|  Nets|    22|
|  Nets|    31|
| Kings|    26|
| Spurs|    40|
|Lakers|    23|
| Spurs|    17|
+------+------+
```

Notice that none of the rows in the resulting DataFrame contain "avs" in the **team** column.

Note that the rows that contained **Mavs** and **Cavs** in the **team** column have both been filtered out since both of these teams contained "avs" in their name.

Note: The **contains** function is case-sensitive. For example, if you would have used "AVS" then the function would not have filtered out the Mavs and Cavs from the DataFrame.

PySpark: How to Filter Using "Contains"

You can use the following syntax to filter a PySpark DataFrame using a "contains" operator:

```
#filter DataFrame where team column contains 'avs'
df.filter(df.team.contains('avs')).show()
```

The following example shows how to use this syntax in practice.

Example: How to Filter Using "Contains" in PySpark

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['Mavs', 14],
        ['Nets', 22],
        ['Nets', 31],
        ['Cavs', 27],
        ['Kings', 26],
```

```
        ['Spurs', 40],
        ['Lakers', 23],
        ['Spurs', 17],]

#define column names
columns = ['team', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+------+------+
|  team|points|
+------+------+
|  Mavs|    14|
|  Nets|    22|
|  Nets|    31|
|  Cavs|    27|
| Kings|    26|
| Spurs|    40|
|Lakers|    23|
| Spurs|    17|
+------+------+
```

We can use the following syntax to filter the DataFrame to only contain rows where the **team** column contains "avs" somewhere in the string:

```
#filter DataFrame where team column contains 'avs'
df.filter(df.team.contains('avs')).show()

+----+------+
|team|points|
+----+------+
|Mavs|    14|
|Cavs|    27|
+----+------+
```

Notice that each of the rows in the resulting DataFrame contain "avs" in the **team** column.

No other rows contained "avs" in the **team** column, which is why all other rows were filtered out of the DataFrame.

Note: The **contains** function is case-sensitive. For example, if you would have used "AVS" then the filter would not have returned any rows because no team name contained "AVS" in all uppercase letters.

PySpark: Filter for Rows that Contain One of Multiple Values

You can use the following syntax to filter for rows in a PySpark DataFrame that contain one of multiple values:

```
#define array of substrings to search for
my_values = ['ets', 'urs']
regex_values = "|".join(my_values)

filter DataFrame where team column contains any substring from
array
df.filter(df.team.rlike(regex_values)).show()
```

The following example shows how to use this syntax in practice.

Example: Filter for Rows that Contain One of Multiple Values in PySpark

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['Mavs', 14],
        ['Nets', 22],
        ['Nets', 31],
        ['Cavs', 27],
        ['Kings', 26],
        ['Spurs', 40],
        ['Lakers', 23],
        ['Spurs', 17],]

#define column names
columns = ['team', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()
```

```
+------+------+
|  team|points|
+------+------+
|  Mavs|    14|
|  Nets|    22|
|  Nets|    31|
|  Cavs|    27|
| Kings|    26|
| Spurs|    40|
|Lakers|    23|
| Spurs|    17|
+------+------+
```

We can use the following syntax to filter the DataFrame to only contain rows where the **team** column contains "ets" or "urs" somewhere in the string:

```
#define array of substrings to search for
my_values = ['ets', 'urs']
regex_values = "|".join(my_values)

filter DataFrame where team column contains any substring from
array
df.filter(df.team.rlike(regex_values)).show()

+-----+------+
| team|points|
+-----+------+
| Nets|    22|
| Nets|    31|
|Spurs|    40|
+-----+------+
```

Notice that each of the rows in the resulting DataFrame contains either "ets" or "urs" in the **team** column

PySpark: How to Filter Rows Based on Values in a List

You can use the following syntax to filter a PySpark DataFrame for rows that contain a value from a specific list:

```
#specify values to filter for
my_list = ['Mavs', 'Kings', 'Spurs']
```

```
#filter for rows where team is in list
df.filter(df.team.isin(my_list)).show()
```

This particular example filters the DataFrame to only contain rows where the value in the **team** column is equal to one of the values in the list that we specified.

The following example shows how to use this syntax in practice.

Example: How to Filter Rows Based on Values in List in PySpark

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['Mavs', 18],
        ['Nets', 33],
        ['Lakers', 12],
        ['Mavs', 15],
        ['Kings', 19],
        ['Wizards', 24],
        ['Magic', 28],
        ['Nets', 40],
        ['Mavs', 24],
        ['Spurs', 13]]

#define column names
columns = ['team', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+-------+------+
|   team|points|
+-------+------+
|   Mavs|    18|
|   Nets|    33|
| Lakers|    12|
|   Mavs|    15|
|  Kings|    19|
```

```
|Wizards|    24|
|  Magic|    28|
|   Nets|    40|
|   Mavs|    24|
|  Spurs|    13|
+-------+------+
```

We can use the following syntax to filter the DataFrame for rows where the **team** column is equal to a team name in a specific list:

```
#specify values to filter for
my_list = ['Mavs', 'Kings', 'Spurs']

#filter for rows where team is in list
df.filter(df.team.isin(my_list)).show()

+-----+------+
| team|points|
+-----+------+
| Mavs|    18|
| Mavs|    15|
|Kings|    19|
| Mavs|    24|
|Spurs|    13|
+-----+------+
```

Notice that each of the rows in the filtered DataFrame have a **team** value equal to either **Mavs**, **Kings** or **Spurs**, which are the three team names that we specified in our list.

Note #1: The **isin** function is case-sensitive.

PySpark: How to Filter Rows Using LIKE Operator

You can use the following syntax to filter a PySpark DataFrame using a LIKE operator:

```
df.filter(df.team.like('%avs%')).show()
```

This particular example filters the DataFrame to only show rows where the string in the **team** column has a pattern like “avs” somewhere in the string.

The following example shows how to use this syntax in practice.

Example: How to Filter Using LIKE Operator in PySpark

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['Mavs', 18],
        ['Nets', 33],
        ['Lakers', 12],
        ['Mavs', 15],
        ['Cavs', 19],
        ['Wizards', 24],
        ['Cavs', 28],
        ['Nets', 40],
        ['Mavs', 24],
        ['Spurs', 13]]

#define column names
columns = ['team', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+-------+------+
|   team|points|
+-------+------+
|   Mavs|    18|
|   Nets|    33|
| Lakers|    12|
|   Mavs|    15|
|   Cavs|    19|
|Wizards|    24|
|   Cavs|    28|
|   Nets|    40|
|   Mavs|    24|
|  Spurs|    13|
+-------+------+
```

We can use the following syntax to filter the DataFrame to only contain rows where the **team** column contains the pattern “avs” somewhere in the string:

```
#filter DataFrame where team column contains pattern like 'avs'
```

```
df.filter(df.team.like('%avs%')).show()

+----+------+
|team|points|
+----+------+
|Mavs|    18|
|Mavs|    15|
|Cavs|    19|
|Cavs|    28|
|Mavs|    24|
+----+------+
```

Notice that each of the rows in the resulting DataFrame contain “avs” in the **team** column.

PySpark: How to Filter Rows Using NOT LIKE

You can use the following syntax to filter a PySpark DataFrame using a NOT LIKE operator:

```
df.filter(~df.team.like('%avs%')).show()
```

This particular example filters the DataFrame to only show rows where the string in the **team** column does not have a pattern like “avs” somewhere in the string.

The following example shows how to use this syntax in practice.

Example: How to Filter Using NOT LIKE in PySpark

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['Mavs', 18],
        ['Nets', 33],
        ['Lakers', 12],
        ['Mavs', 15],
        ['Cavs', 19],
        ['Wizards', 24],
        ['Cavs', 28],
        ['Nets', 40],
```

```
        ['Mavs', 24],
        ['Spurs', 13]]

#define column names
columns = ['team', 'points']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+-------+------+
|   team|points|
+-------+------+
|   Mavs|    18|
|   Nets|    33|
| Lakers|    12|
|   Mavs|    15|
|   Cavs|    19|
|Wizards|    24|
|   Cavs|    28|
|   Nets|    40|
|   Mavs|    24|
|  Spurs|    13|
+-------+------+
```

We can use the following syntax to filter the DataFrame to only contain rows where the **team** column does not contain a pattern like "avs" somewhere in the string:

```
#filter DataFrame where team column does not contain pattern like
'avs'
df.filter(~df.team.like('%avs%')).show()

+-------+------+
|   team|points|
+-------+------+
|   Nets|    33|
| Lakers|    12|
|Wizards|    24|
|   Nets|    40|
|  Spurs|    13|
+-------+------+
```

Notice that each of the rows in the resulting DataFrame do not contain a pattern like "avs" in the **team** column.

Note that we used the **like** function to find all strings in the team column that had a pattern like "avs" and then we used the ~ symbol to negate this function.

The end result is that we're able to filter for only the rows in the DataFrame that do not have a pattern like "avs" in the team column.

How to Filter by Date Range in PySpark (With Example)

You can use the following syntax to filter rows in a PySpark DataFrame based on a date range:

```
#specify start and end dates
dates = ('2019-01-01', '2022-01-01')

#filter DataFrame to only show rows between start and end dates
df.filter(df.start_date.between(*dates)).show()
```

This particular example filters the DataFrame to only contain rows where the date in the **start_date** column of the DataFrame is between **2019-01-01** and **2022-01-01**.

The following example shows how to use this syntax in practice.

Example: How to Filter by Date Range in PySpark

Suppose we have the following PySpark DataFrame that contains information about the start date for various employees at a company:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', '2017-10-25'],
        ['B', '2018-10-11'],
        ['C', '2018-10-17'],
        ['D', '2019-12-21'],
        ['E', '2021-04-14'],
        ['F', '2022-06-26']]

#define column names
columns = ['employee', 'start_date']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()
```

```
+--------+----------+
|employee|start_date|
+--------+----------+
|       A|2017-10-25|
|       B|2018-10-11|
|       C|2018-10-17|
|       D|2019-12-21|
|       E|2021-04-14|
|       F|2022-06-26|
+--------+----------+
```

We can use the following syntax to filter the DataFrame to only contain rows where the date in the **start_date** column of the DataFrame is between **2019-01-01** and **2022-01-01**:

```
#specify start and end dates
dates = ('2019-01-01', '2022-01-01')

#filter DataFrame to only show rows between start and end dates
df.filter(df.start_date.between(*dates)).show()

+--------+----------+
|employee|start_date|
+--------+----------+
|       D|2019-12-21|
|       E|2021-04-14|
+--------+----------+
```

Notice that the DataFrame has been filtered to only show the rows with the two dates in the **start_date** column that fall between **2019-01-01** and **2022-01-01**.

Note that if you only want to know how many rows have a date within a specific date range, then you can use the **count** function as follows:

```
#specify start and end dates
dates = ('2019-01-01', '2022-01-01')

#count number of rows in DataFrame that fall between start and end
dates
df.filter(df.start_date.between(*dates)).count()

2
```

This tells us that there are two rows in the DataFrame where the date in the **start_date** column falls between **2019-01-01** and **2022-01-01**.

PySpark: How to Filter by Boolean Column

You can use the following methods to filter the rows of a PySpark DataFrame based on values in a Boolean column:

Method 1: Filter Based on Values in One Boolean Column

```
#filter for rows where value in 'all_star' column is True
df.filter(df.all_star==True).show()
```

Method 2: Filter Based on Values in Multiple Boolean Columns

```
#filter for rows where value in 'all_star' and 'starter' columns
are both True
df.filter((df.all_star==True) & (df.starter==True)).show()
```

The following examples show how to use each method in practice with the following PySpark DataFrame that contains information about various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 18, True, False],
        ['B', 20, False, True],
        ['C', 25, True, True],
        ['D', 40, True, True],
        ['E', 34, True, False],
        ['F', 32, False, False],
        ['G', 19, False, False]]

#define column names
columns = ['team', 'points', 'all_star', 'starter']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+------+--------+-------+
|team|points|all_star|starter|
+----+------+--------+-------+
|   A|    18|    true|  false|
|   B|    20|   false|   true|
|   C|    25|    true|   true|
|   D|    40|    true|   true|
|   E|    34|    true|  false|
```

```
|   F|    32|   false|  false|
|   G|    19|   false|  false|
+----+------+--------+-------+
```

Example 1: Filter Based on Values in One Boolean Column

We can use the following syntax to filter the DataFrame to only contain rows where the value in the **all_star** column is true:

```
#filter for rows where value in 'all_star' column is True
df.filter(df.all_star==True).show()

+----+------+--------+-------+
|team|points|all_star|starter|
+----+------+--------+-------+
|   A|    18|    true|  false|
|   C|    25|    true|   true|
|   D|    40|    true|   true|
|   E|    34|    true|  false|
+----+------+--------+-------+
```

Notice that each of the rows in the filtered DataFrame have a value of **true** in the **all_star** column.

Example 2: Filter Based on Values in Multiple Boolean Columns

We can use the following syntax to filter the DataFrame to only contain rows where the value in the **all_star** column is true *and* the value in the **starter** column is true:

```
#filter for rows where value in 'all_star' and 'starter' columns
are both True
df.filter((df.all_star==True) & (df.starter==True)).show()

+----+------+--------+-------+
|team|points|all_star|starter|
+----+------+--------+-------+
|   C|    25|    true|   true|
|   D|    40|    true|   true|
+----+------+--------+-------+
```

Notice that each of the rows in the filtered DataFrame have a value of **true** in both the **all_star** and **starter** columns.

PySpark: How to Use fillna() with Specific Columns

You can use the following methods with **fillna()** to replace null values in specific columns of a PySpark DataFrame:

Method 1: Use fillna() with One Specific Column

```
df.fillna(0, subset='col1').show()
```

Method 2: Use fillna() with Several Specific Columns

```
df.fillna(0, subset=['col1', 'col2']).show()
```

The following examples show how to use each method in practice with the following PySpark DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['A', 'East', 11, 4],
        ['A', 'East', 8, 9],
        ['A', 'East', None, 3],
        ['B', 'West', None, 12],
        ['B', 'West', 6, 4],
        ['C', None, 5, None]]

#define column names
columns = ['team', 'conference', 'points', 'assists']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|  null|      3|
|   B|      West|  null|     12|
```

```
|   B|      West|     6|      4|
|   C|      null|     5|   null|
+----+----------+------+-------+
```

Example 1: Use fillna() with One Specific Column

We can use the following syntax to fill the null values in the **points** column only with zeros:

```
#fill null values in 'points' column with zeros
df.fillna(0, subset='points').show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|     0|      3|
|   B|      West|     0|     12|
|   B|      West|     6|      4|
|   C|      null|     5|   null|
+----+----------+------+-------+
```

Notice that each of the null values in the **points** column have been replaced with zeros while the null values in all other columns have been left unchanged.

Example 2: Use fillna() with Several Specific Columns

We can use the following syntax to fill the null values in the **points** and **assists** columns with zeros:

```
#fill null values in 'points' and 'assists' column with zeros
df.fillna(0, subset=['points', 'assists']).show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|   A|      East|    11|      4|
|   A|      East|     8|      9|
|   A|      East|     0|      3|
|   B|      West|     0|     12|
|   B|      West|     6|      4|
|   C|      null|     5|      0|
+----+----------+------+-------+
```

Notice that each of the null values in the **points** and **assists** columns have been replaced with zeros while the null values in all other columns have been left unchanged.

Note #1: We chose to replace the null values with 0 but you can use any value you'd like as a replacement.

PySpark: How to Use fillna() with Another Column

You can use the following syntax with **fillna()** to replace null values in one column with corresponding values from another column in a PySpark DataFrame:

```
from pyspark.sql.functions import coalesce

df.withColumn('points', coalesce('points',
'points_estimate')).show()
```

This particular example replaces null values in the **points** column with corresponding values from the **points_estimate** column.

The following example shows how to use this syntax in practice.

Example: How to Use fillna() with Another Column in PySpark

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = [['Mavs', 18, 18],
        ['Nets', 33, 33],
        ['Lakers', None, 25],
        ['Kings', 15, 15],
        ['Hawks', None, 29],
        ['Wizards', None, 14],
        ['Magic', 28, 28]]

#define column names
columns = ['team', 'points', 'points_estimate']

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)
```

```
#view dataframe
df.show()

+-------+------+---------------+
|   team|points|points_estimate|
+-------+------+---------------+
|   Mavs|    18|             18|
|   Nets|    33|             33|
| Lakers|  null|             25|
|  Kings|    15|             15|
|  Hawks|  null|             29|
|Wizards|  null|             14|
|  Magic|    28|             28|
+-------+------+---------------+
```

Suppose we would like to fill in all of the null values in the **points** column with corresponding values from the **points_estimate** column.

We can use the following syntax to do so:

```
from pyspark.sql.functions import coalesce

#replace null values in 'points' column with values from
'points_estimate' column
df.withColumn('points', coalesce('points',
'points_estimate')).show()

+-------+------+---------------+
|   team|points|points_estimate|
+-------+------+---------------+
|   Mavs|    18|             18|
|   Nets|    33|             33|
| Lakers|    25|             25|
|  Kings|    15|             15|
|  Hawks|    29|             29|
|Wizards|    14|             14|
|  Magic|    28|             28|
+-------+------+---------------+
```

Notice that each of the null values in the **points** column have been replaced with the corresponding values from the **points_estimate** column.