[1 SparkContext 5](#_Toc4511149)

[1.1 RDD 5](#_Toc4511150)

[1.1.1 Load data 6](#_Toc4511151)

[1.1.2 Read 6](#_Toc4511152)

[1.1.3 Partitions 6](#_Toc4511153)

[1.1.4 Transformations 7](#_Toc4511154)

[1.1.5 Actions 9](#_Toc4511155)

[1.2 Pair RDD 10](#_Toc4511156)

[1.2.1 Load data 10](#_Toc4511157)

[1.2.2 Transformations 11](#_Toc4511158)

[1.2.3 Actions 12](#_Toc4511159)

[1.3 Word Count 14](#_Toc4511160)

[2 SparkSession 15](#_Toc4511161)

[2.1 Tables 15](#_Toc4511162)

[2.2 SQL 15](#_Toc4511163)

[2.3 Pandafy 16](#_Toc4511164)

[2.4 Sparkify 16](#_Toc4511165)

[3 Spark dataframes 18](#_Toc4511166)

[3.1 Create dataframe from RDD 18](#_Toc4511167)

[3.2 Read 19](#_Toc4511168)

[3.3 Create columns 19](#_Toc4511169)

[3.4 Transformations 20](#_Toc4511170)

[3.4.1 Rename columns 20](#_Toc4511171)

[3.4.2 Distinct 20](#_Toc4511172)

[3.4.3 Where 20](#_Toc4511173)

[3.4.4 Isin 20](#_Toc4511174)

[3.4.5 Filter rows 21](#_Toc4511175)

[3.4.6 Select rows 21](#_Toc4511176)

[3.4.7 Select columns 21](#_Toc4511177)

[3.4.8 Alias 21](#_Toc4511178)

[3.4.9 SelectExpr 22](#_Toc4511179)

[3.4.10 GroupBy 22](#_Toc4511180)

[3.4.11 Agg 23](#_Toc4511181)

[3.4.12 OrderBy 23](#_Toc4511182)

[3.4.13 Drop columns 24](#_Toc4511183)

[3.4.14 DropDuplicates 24](#_Toc4511184)

[3.4.15 Join 25](#_Toc4511185)

[3.4.16 Text filtering 26](#_Toc4511186)

[3.4.17 Value filtering 26](#_Toc4511187)

[3.5 Actions 26](#_Toc4511188)

[3.5.1 PrintSchema 26](#_Toc4511189)

[3.5.2 Head 27](#_Toc4511190)

[3.5.3 Show 27](#_Toc4511191)

[3.5.4 Count 27](#_Toc4511192)

[3.5.5 Columns 27](#_Toc4511193)

[3.5.6 Describe 27](#_Toc4511194)

[4 ML 28](#_Toc4511195)

[4.1 Dtypes 28](#_Toc4511196)

[4.2 Cast 28](#_Toc4511197)

[4.3 Count NA 28](#_Toc4511198)

[4.4 Drop NA 28](#_Toc4511199)

[4.5 Fill NA 28](#_Toc4511200)

[4.6 Not 29](#_Toc4511201)

[4.7 Descriptive functions 29](#_Toc4511202)

[4.8 Scaling 30](#_Toc4511203)

[4.8.1 Minmax 30](#_Toc4511204)

[4.8.2 Standardization 30](#_Toc4511205)

[4.8.3 Log transformation 30](#_Toc4511206)

[4.8.4 Reflecting 30](#_Toc4511207)

[4.9 Encoding 31](#_Toc4511208)

[4.10 Assembler 32](#_Toc4511209)

[4.11 Pipeline 32](#_Toc4511210)

[4.12 Split 32](#_Toc4511211)

[4.13 Model 32](#_Toc4511212)

[4.14 Train 33](#_Toc4511213)

[4.15 Cross validate 33](#_Toc4511214)

[4.16 Evaluate 34](#_Toc4511215)

[5 MLLib 34](#_Toc4511216)

[5.1 Vectors 35](#_Toc4511217)

[5.2 LabelledPoint 35](#_Toc4511218)

[5.3 HashingTF 35](#_Toc4511219)

[5.4 Recommendation 36](#_Toc4511220)

[5.4.1 MovieLens 37](#_Toc4511221)

[5.5 Classification 38](#_Toc4511222)

[5.5.1 Spam 38](#_Toc4511223)

[5.6 Clustering 39](#_Toc4511224)

[5.6.1 5000 points 40](#_Toc4511225)

[6 Data Visualization 41](#_Toc4511226)

[6.1 pyspark\_dist\_explore 41](#_Toc4511227)

[6.2 ToPandas 42](#_Toc4511228)

[6.3 HandySpark 42](#_Toc4511229)

[7 Feature engineering 42](#_Toc4511230)

[7.1 Parquet 42](#_Toc4511231)

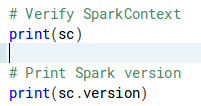
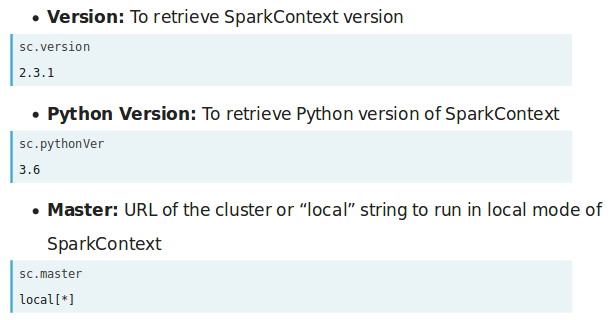
[7.2 Date 43](#_Toc4511232)

[7.2.1 Time components 44](#_Toc4511233)

[7.2.2 Diff 44](#_Toc4511234)

[7.2.3 Lag 44](#_Toc4511235)

# SparkContext

* Creating the connection is as simple as creating an instance of the SparkContext class. The class constructor takes a few optional arguments that allow you to specify the attributes of the cluster you're connecting to.
* An object holding all these attributes can be created with the SparkConf() constructor. Take a look at the [documentation](http://spark.apache.org/docs/2.1.0/api/python/pyspark.html" \t "/home/kaushik/Documentsx/_blank) for all the details!
* 
* 
  + The “\*” above indicates that the master is capable of accessing all available threads

## RDD

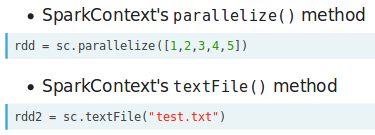
### Challenges

* Spark does not know the inner structures of user-defined RDD records, as it does for user-defined dataframes - therefore all the optimization automatically available with dataframes need to be manually re-created for RDDs
* One can get good performance with RDDs running on scala or Java, but running Python RDDs, like PySpark UDFs, require serialization of data from JVM to PySpark process through Py4J, work on it in Python, and then serialize it back the JVM - and therefore Python RDDs should be used only when absolutely necessary
* For the performance comparison, please visit <https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html>
* RDDs don’t have access to built-in functions - which means we need to define each filter, map and aggregation as a function

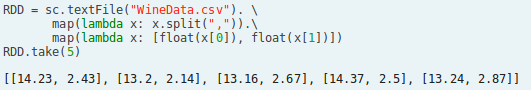
### Advantages

* Fine-grained control over the physical distribution and partitioning of the data

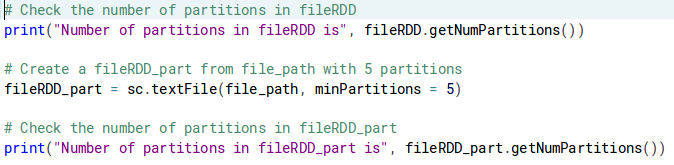
### Load data

* SparkContext is the main entry point for creating RDDs
* 
* 

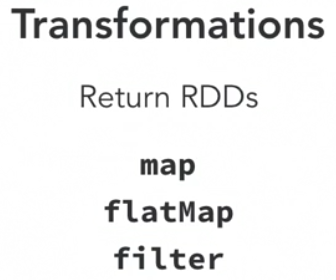
### Read

* 

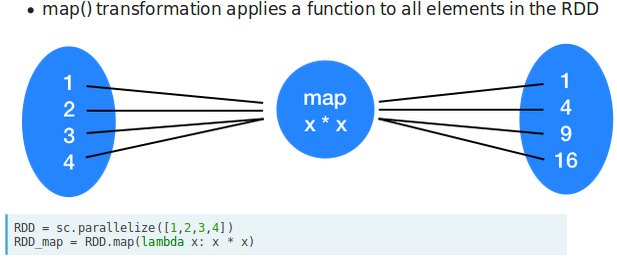
### Partitions

* 

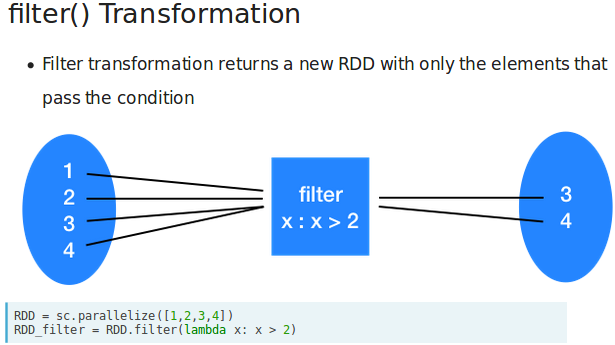
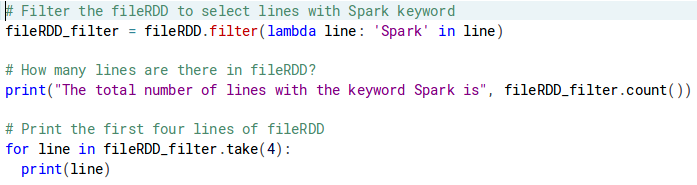
### Transformations

* 

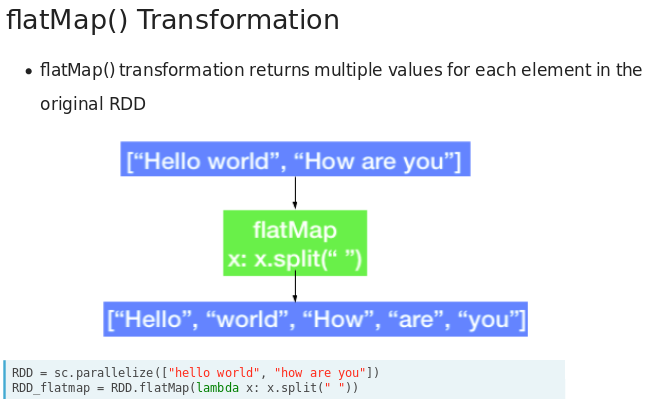
#### Map

* 

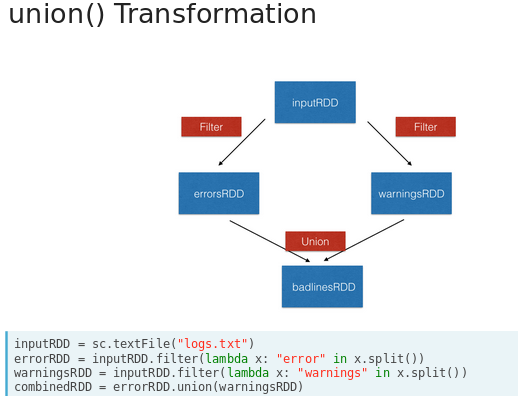
#### Filter

* 
* Note that the filter() operation does not mutate the existing fileRDD. Instead, it returns a pointer to an entirely new RDD.
* 

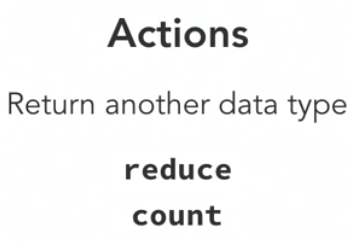
#### FlatMap

* 

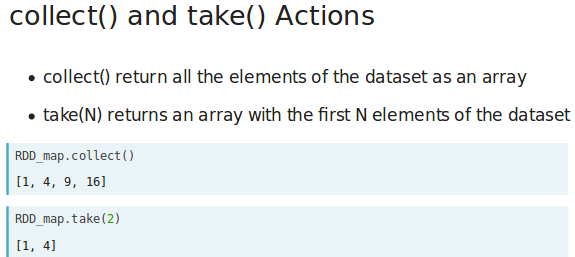
#### Union

* 

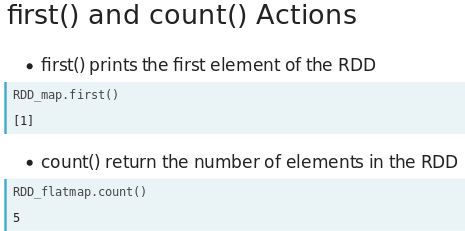
### Actions

* 

#### Collect and Take

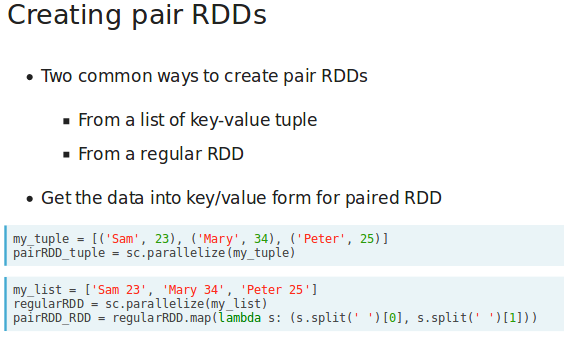
* 

#### First and Count

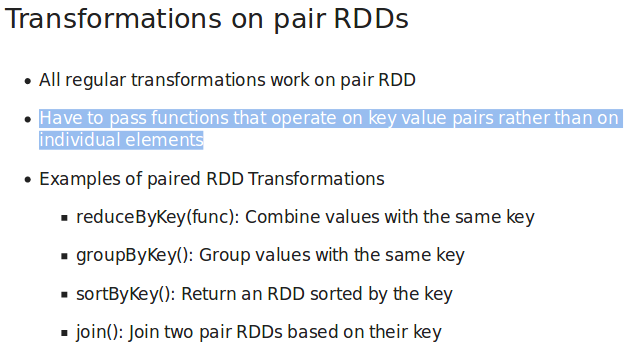
* 

## Pair RDD

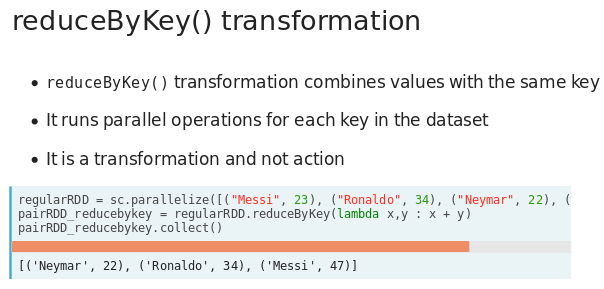
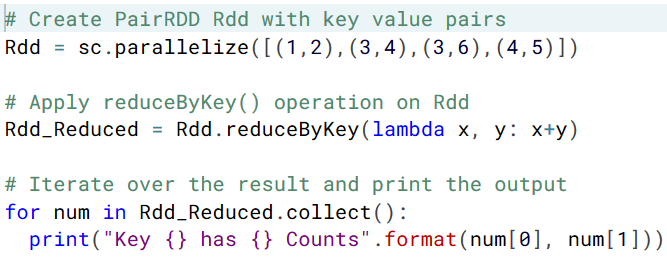
### Load data

* 

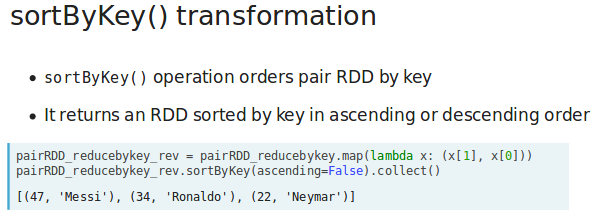
### Transformations

* 

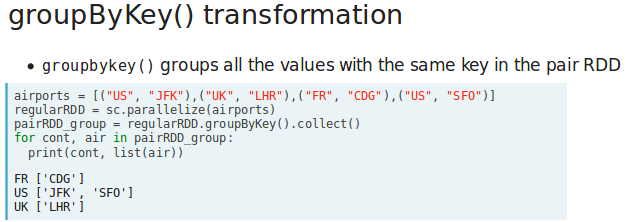
#### ReduceByKey

* 
* 

#### SortByKey

* 

#### GroupByKey

* 

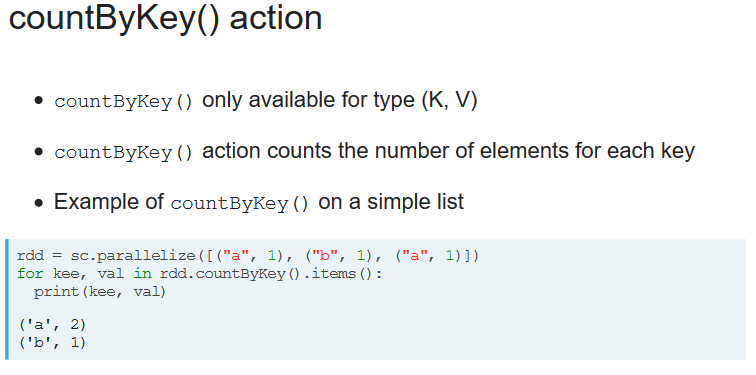
#### Join

* 

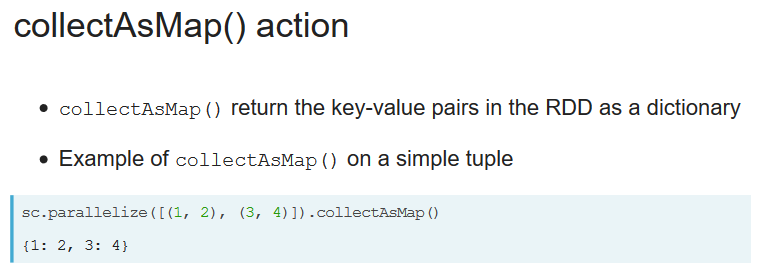
### Actions

* All the RDD actions are available for PySpark pair RDDs also
* But in addition, pair RDD actions can leverage the key-value data
* Few examples of pair RDD actions that can be run only on dataset small enough to fit inside memory are:
  + countByKey()
  + collectAsMap()

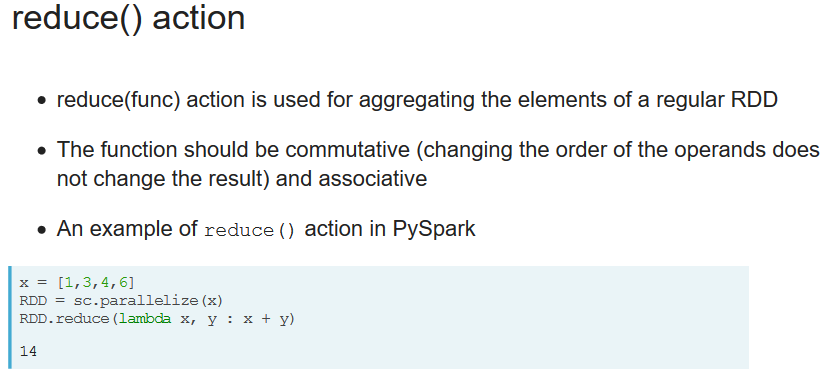
#### CounByKey

* 

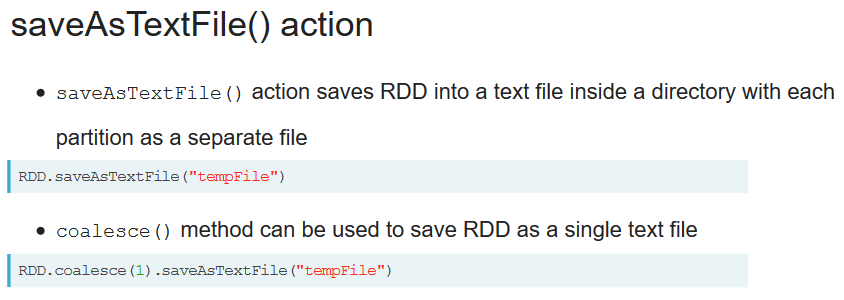
#### CollectAsMap

* 

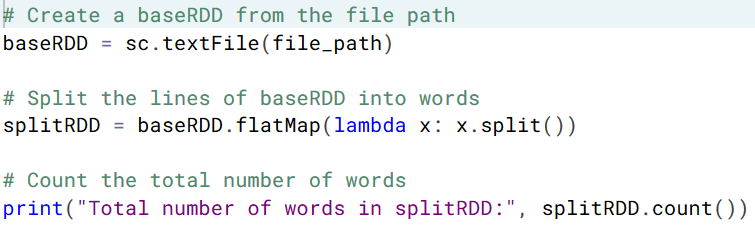
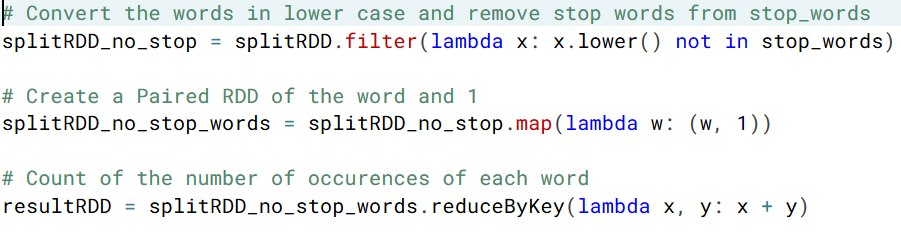
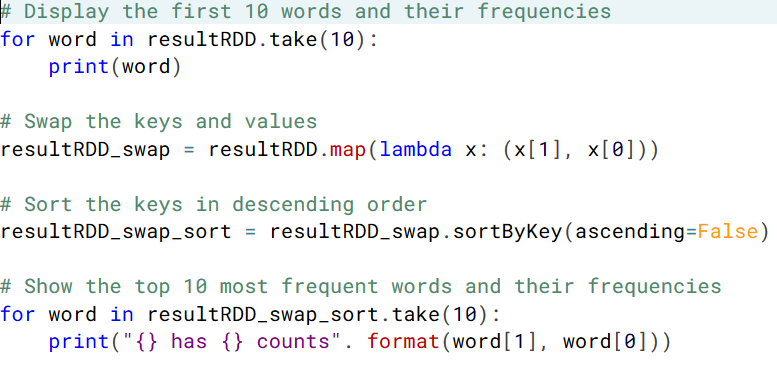
#### Reduce

* 

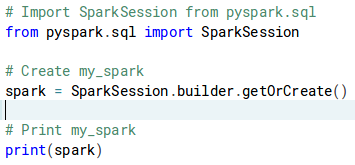
#### SaveAsTextFile

* 

## Word Count

* 
* 
* 

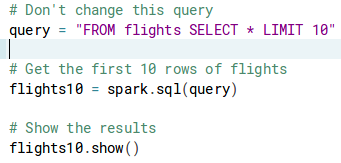
# SparkSession

* To start working with Spark DataFrames, you first have to create a SparkSession object from your SparkContext. You can think of the SparkContext as your connection to the cluster and the SparkSession as your interface with that connection.
* 

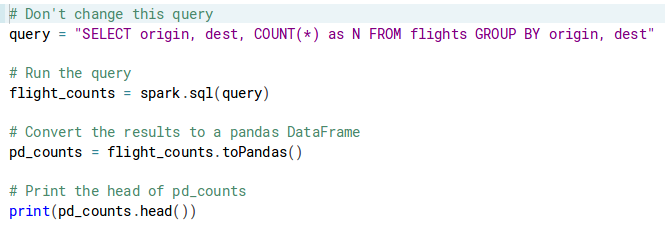
## Tables

* Your SparkSession has an attribute called catalog which lists all the data inside the cluster.
* This attribute has a few methods for extracting different pieces of information. One of the most useful is the .listTables() method, which returns the names of all the tables in your cluster as a list.
* 

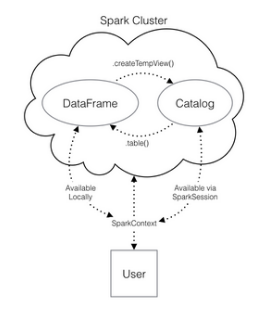
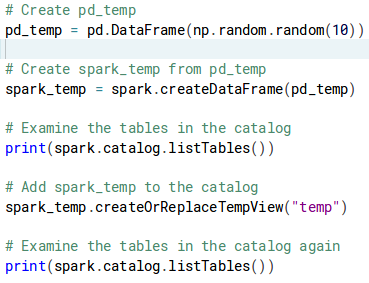
## SQL

* 
* 
  + If you look closely, you'll notice that the table flights is only mentioned in the query, not as an argument to any of the methods. This is because there isn't a local object in your environment that holds that data, so it wouldn't make sense to pass the table as an argument.

## Pandafy

* Sometimes it makes sense to then take that table and work with it locally using a tool like pandas. Spark DataFrames make that easy with the .toPandas() method. Calling this method on a Spark DataFrame returns the corresponding pandas DataFrame.
* 

## Sparkify

* The .createDataFrame() method takes a pandas DataFrame and returns a Spark DataFrame.
* The output of this method is stored locally, not in the SparkSession catalog. This means that you can use all the Spark DataFrame methods on it, but in order to access the data in other contexts, you have to save it as a temporary table.
  + You can do this using the .createTempView() Spark DataFrame method, which takes as its only argument the name of the temporary table you'd like to register. This method registers the DataFrame as a table in the catalog, but as this table is temporary, it can only be accessed from the specific SparkSession used to create the Spark DataFrame.
  + There is also the method .createOrReplaceTempView(). This safely creates a new temporary table if nothing was there before, or updates an existing table if one was already defined. You'll use this method to avoid running into problems with duplicate tables.
* 
* 

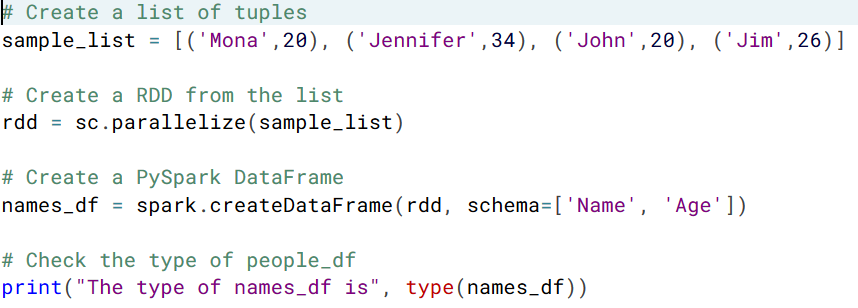
# Spark Datasets

* Datasets are used with statically typed languages such as Java or Scala
* Python being a dynamic language doesn’t support Dataset API
* However, most of the benefits offered by Dataset APIs are already available in Dataframes

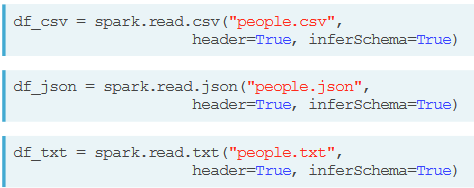
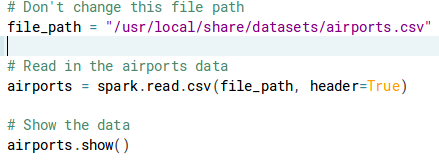
# Spark dataframes

* SparkSession provides a single point of entry to interact with Spark DataFrames

## Create dataframe from RDD

* 
* 

## Read

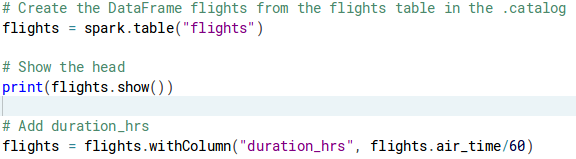
* 
* 

## Transformations

## Add rows (union)

* 

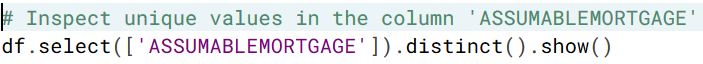
## Add columns

* Updating a Spark DataFrame is somewhat different than working in pandas because the Spark DataFrame is immutable. This means that it can't be changed, and so columns can't be updated in place.
* Thus, all these methods return a new DataFrame. To overwrite the original DataFrame you must reassign the returned DataFrame using the method like:
  + df = df.withColumn("newCol", df.oldCol + 1)
* To overwrite an existing column, just pass the name of the column as the first argument!
* 

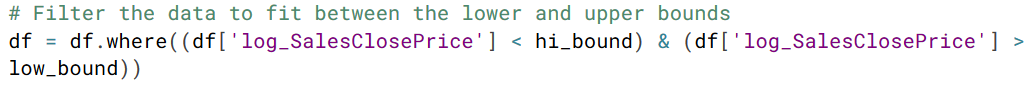
### Rename columns

* 

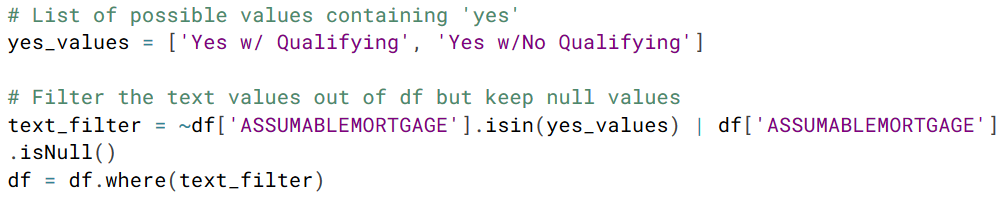
### Distinct

* 

### Where

* 

### Isin

* 

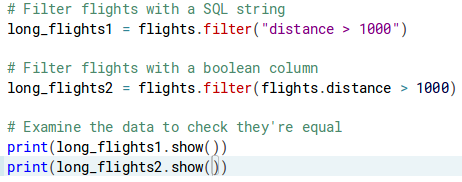
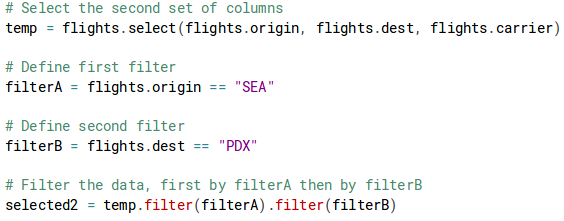
### Select rows

* 

### Select columns

* 
* select1.columns
* 

### Filter rows

* 
* 
* 

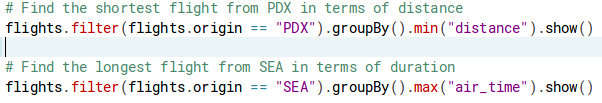
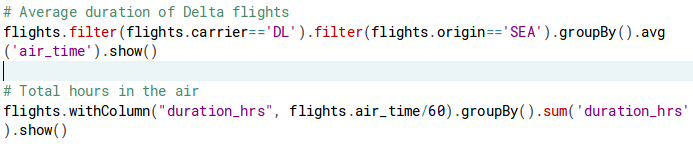
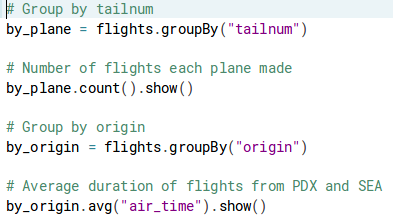
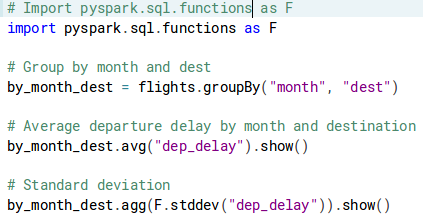
### Alias

* 
* 

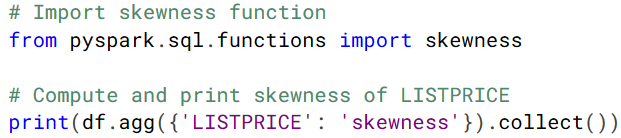
### SelectExpr

* 
* 

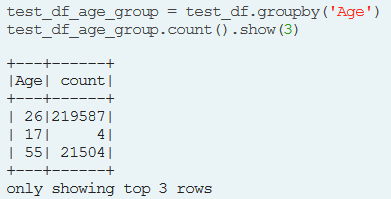
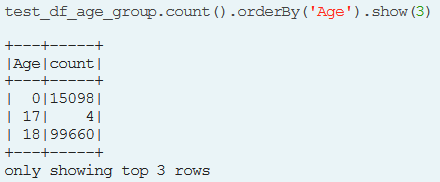
### GroupBy

* 
* 
* 
* 
* 

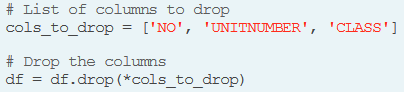
### Agg

* 

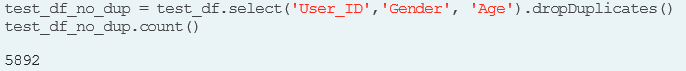
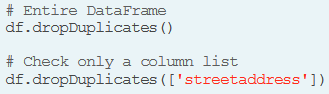
### OrderBy (sort)

* orderby() operation sorts the DataFrame based one or more columns
* 
* 
* 

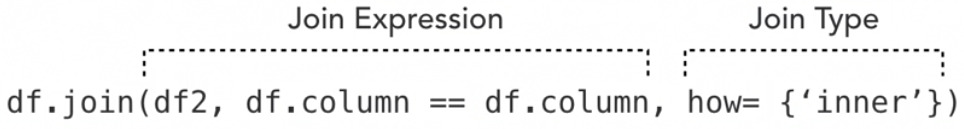
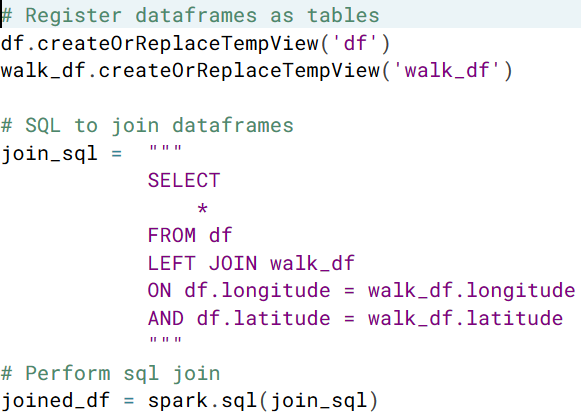
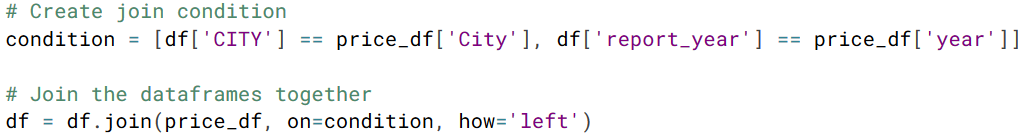
### Drop columns

* 

### DropDuplicates

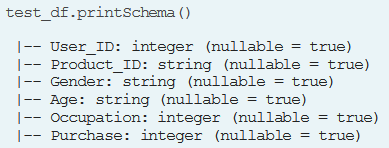
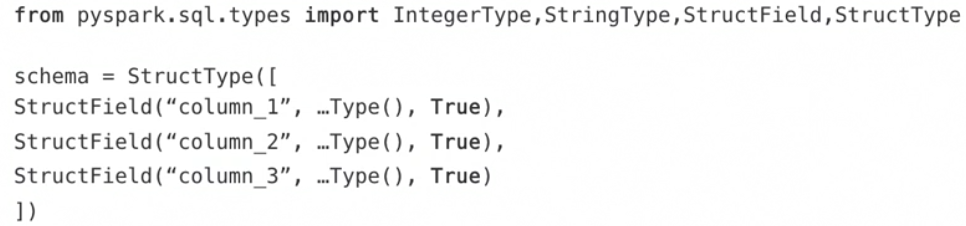
* dropDuplicates() removes the duplicate rows of a DataFrame
* 
* 

### Join

* 
* 
* 
* 

## Actions

### PrintSchema

* printSchema() operation prints the types of columns in the DataFrame
* 
* Explicitly defining the schema in Spark is recommended
* 
  + A schema is a StructType made up of a number of fields of StructField:
    - Column name
    - Type of the column
    - Whether the column can contain missing or null values

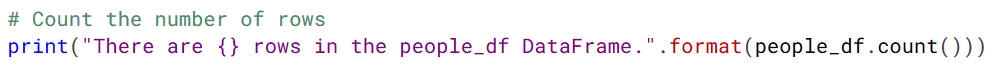
### Head

### Limit

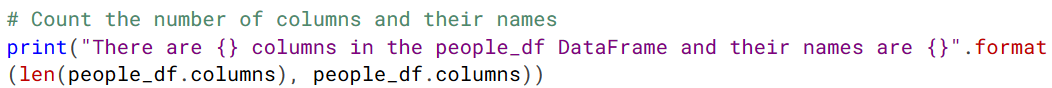
### Show

* 

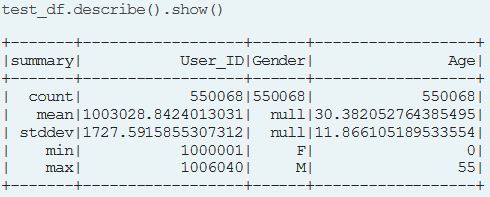
### Count

* 

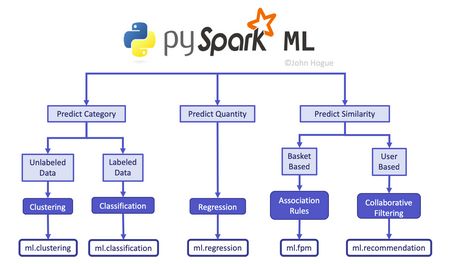
### Columns

* columns() operator returns the columns of a DataFrame as an array of strings
* 
* 

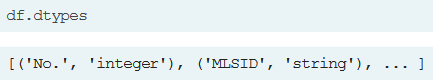
### Describe

* describe() operation compute summary statistics of numerical columns in the DataFrame
* 
* 

# ML

* 

## Dtypes

* 

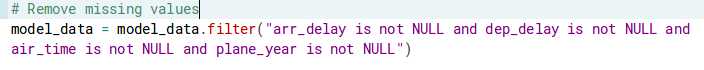
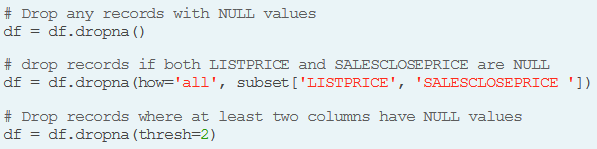
## Cast

* 

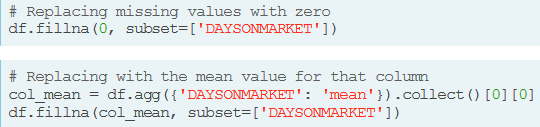
## Count NA

* 

## Drop NA

* 
* 

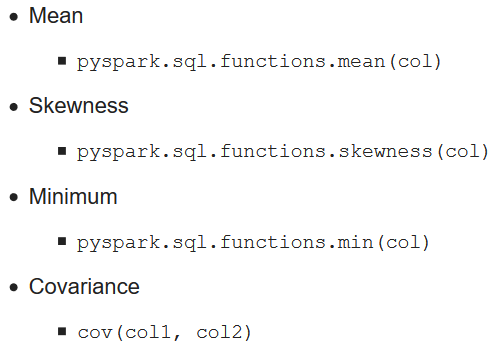
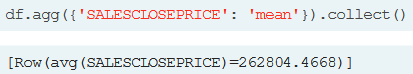
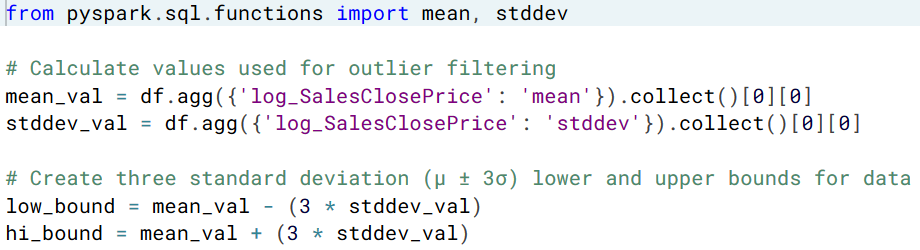
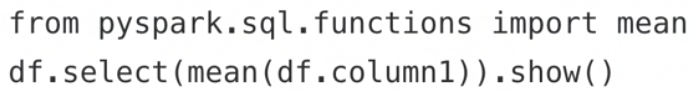
## Fill NA

* 
* 

## Not

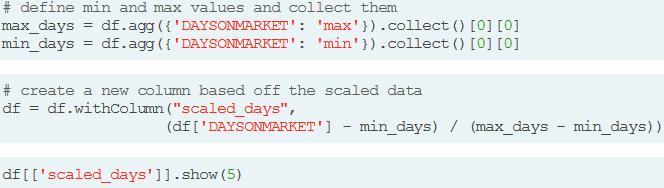
* 

## Built-in functions

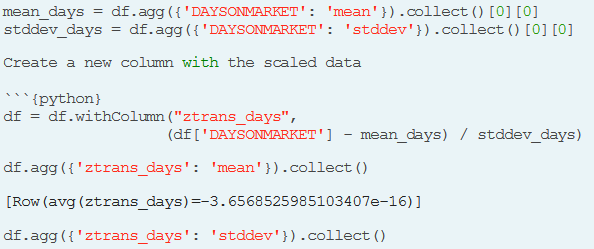
* 
* 
* Correlation
  + corr(col1, col2)
* 
* 
* 
* 

## Scaling

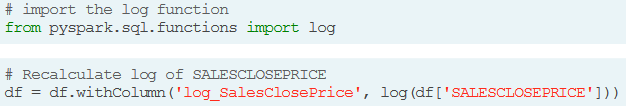
### Minmax

* 

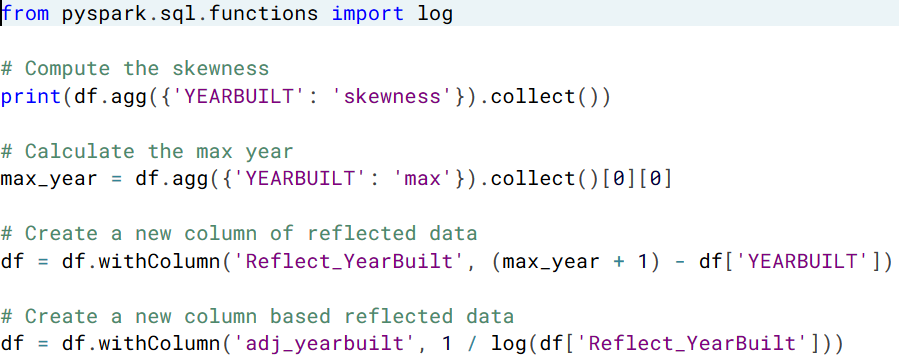
### Standardization

* 

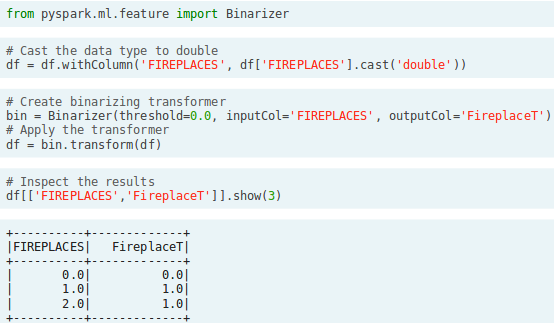
### Log transformation

* 

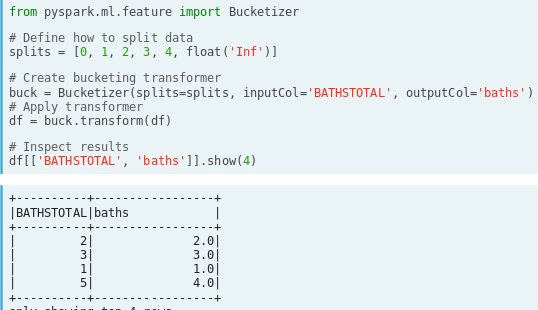
### Reflecting

* You might use log transforms to fix positively skewed data (data whose distribution is mostly to the left). To correct negative skew (data mostly to the right) you need to take an extra step called "reflecting" before you can apply the inverse of log, written as (1/log) to make the data look more like normal a normal distribution. Reflecting data uses the following formula to reflect each value: (*x*max+1)–*x*
* 

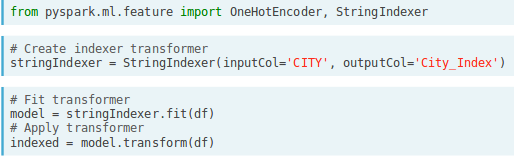
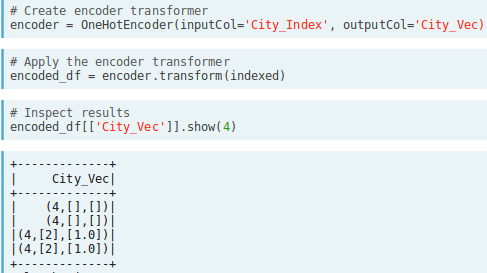
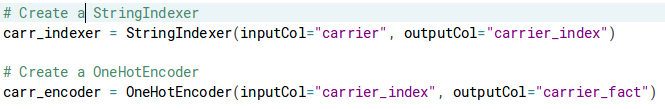
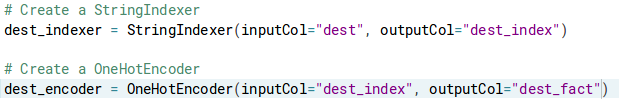
## Binarizing

* 

## Bucketing

* 

## Encoding

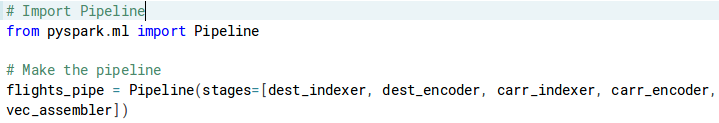
* The first step to encoding your categorical feature is to create a StringIndexer. Members of this class are Estimators that take a DataFrame with a column of strings and map each unique string to a number. Then, the Estimator returns a Transformer that takes a DataFrame, attaches the mapping to it as metadata, and returns a new DataFrame with a numeric column corresponding to the string column.
  + In Spark it's important to make sure you split the data after all the transformations. This is because operations like StringIndexer don't always produce the same index even when given the same list of strings.
* The second step is to encode this numeric column as a one-hot vector using a OneHotEncoder. This works exactly the same way as the StringIndexer by creating an Estimator and then a Transformer. The end result is a column that encodes your categorical feature as a vector that's suitable for machine learning routines!
* Please note that the last category is not included by default, because it is linearly dependent on the other columns and therefore is not needed.
* For Random Forest, categorical features only need to be mapped to numbers, they are fine to stay all in one column by using a StringIndexer as we saw in chapter 3. OneHot encoding which converts each possible value to its own boolean feature is not needed.
* Likewise, Missing values are handled by Random Forests internally where they partition on missing values. As long as you replace them with something outside of the range of normal values, they will be handled correctly.
* 
* 
* 
* 
* It is a good practice to remove columns that have less than 30 observations. 30 is a common minimum number of observations for statistical significance. Any less than that and the relationships cause overfitting because of a sheer coincidence!

## Classification

### Assembler

* pyspark.ml.feature submodule contains a class called VectorAssembler. This Transformer takes all of the columns you specify and combines them into a new vector column.
* 

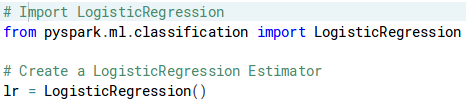
### Pipeline

* 
* 

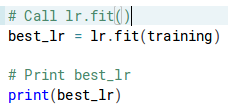
### Split

* 

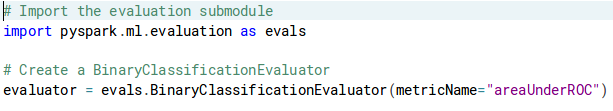
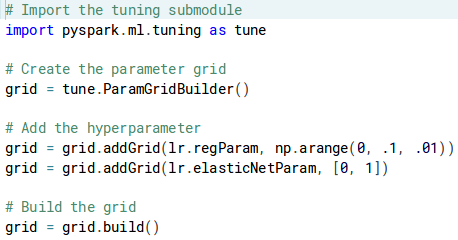
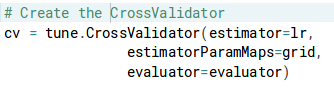
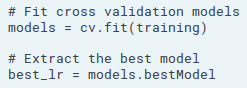
### Model

* 

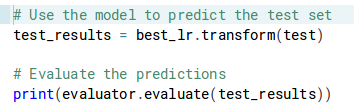
### Train

* 

### Cross validate

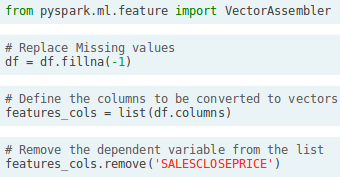
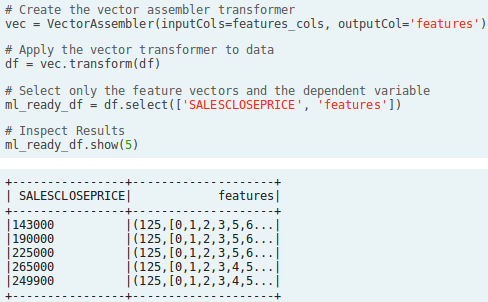
* 
* 
* 
* 

### Evaluate

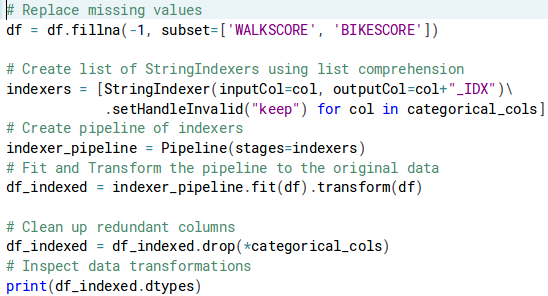
* 

## Regression

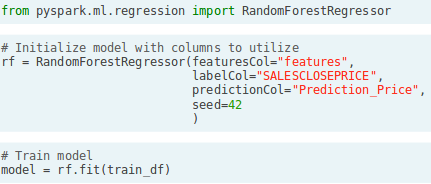
### Assembler

* 
* 

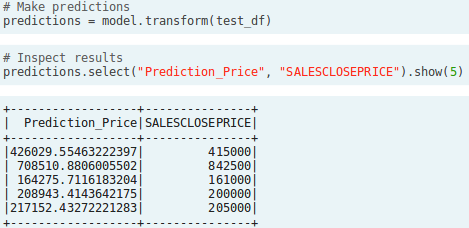
### Pipeline

* 

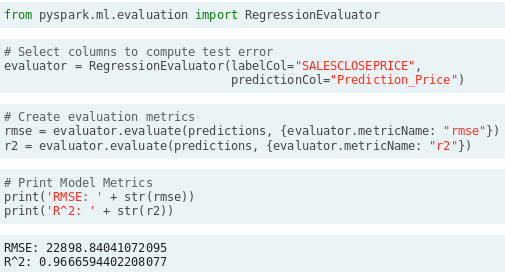
### Model

* 

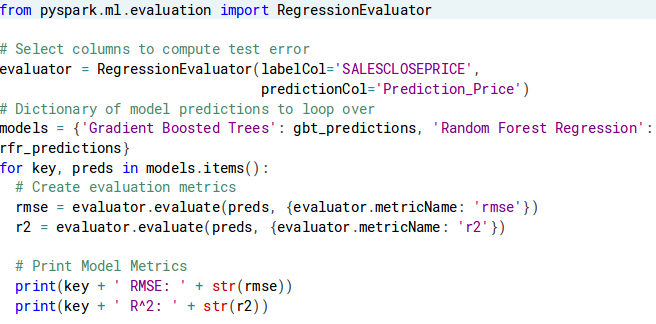
### Predict

* 

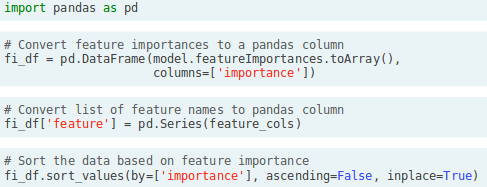
### Evaluate

* 

#### Compare

* 

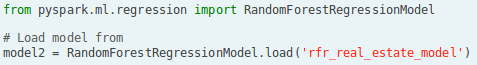
### Interpret model (Feature Importance)

* 
* 

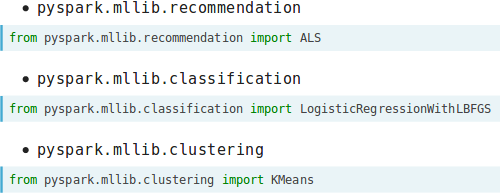
### Save model

* 

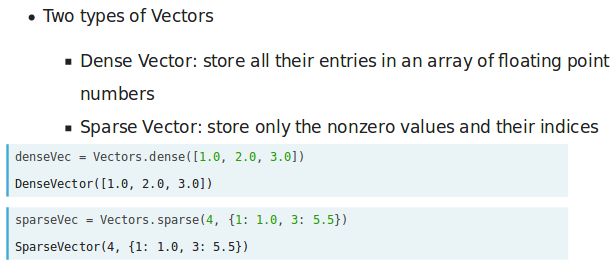
### Load model

* 

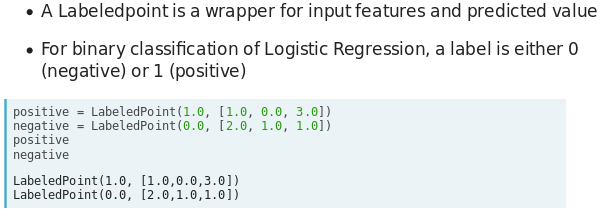
# MLLib

* 

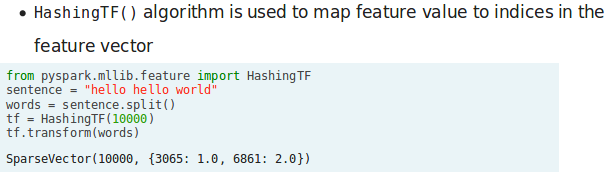
## Vectors

* 

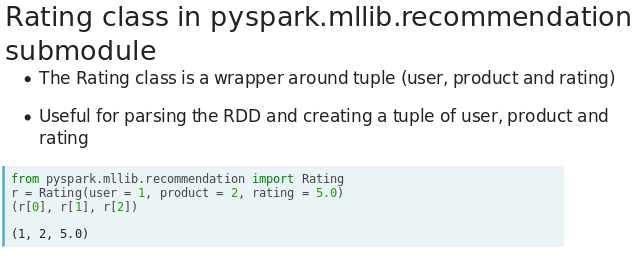
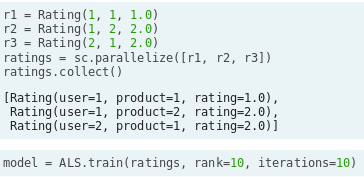
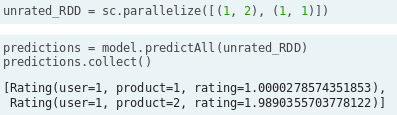
## LabelledPoint

* 

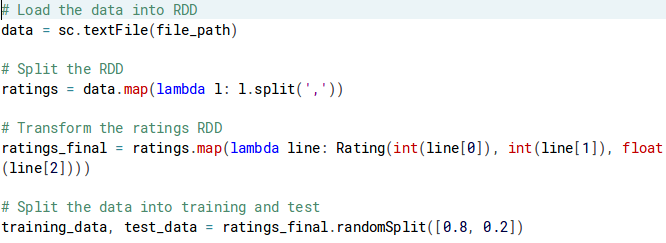
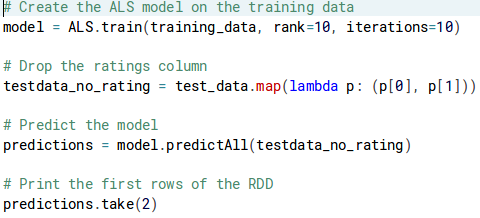
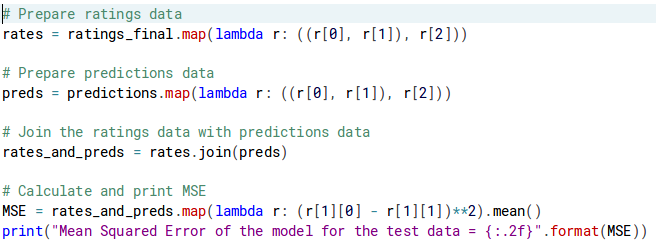
## HashingTF

* 

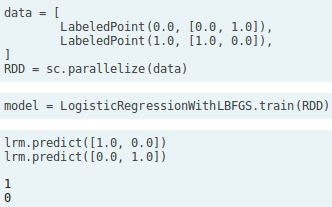
## Recommendation

* PySpark MLlib's ALS algorithm has the following mandatory parameters - rank (the number of latent factors in the model) and iterations (number of iterations to run)
* 
* 
* 
* 

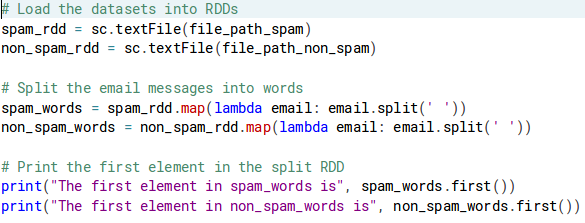
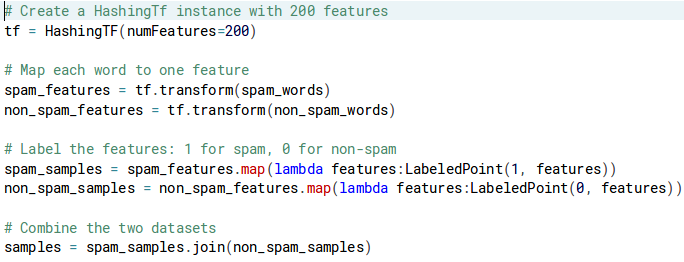
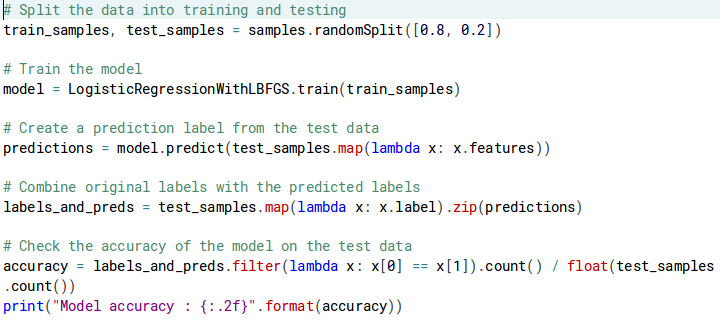
### MovieLens

* 
* 
* 

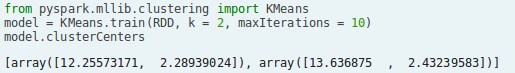
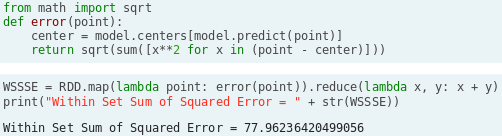
## Classification

* 

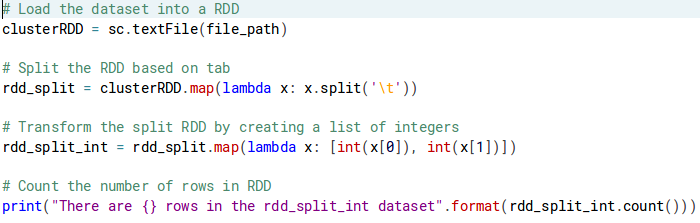
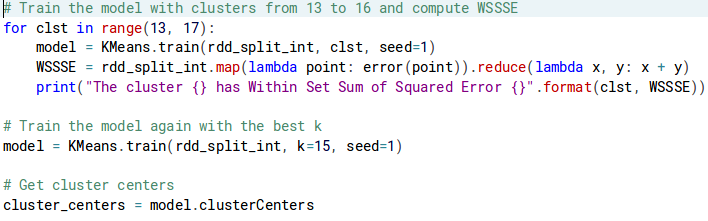
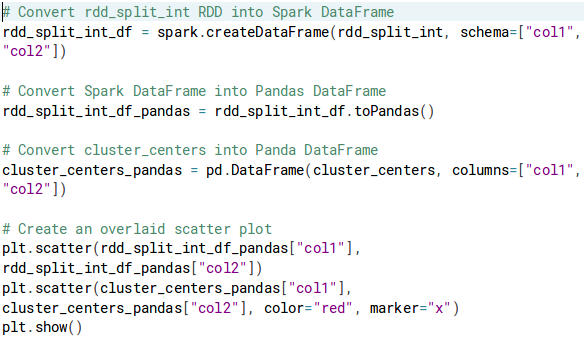
### Spam

* 
* 
* 

## Clustering

* 
* 
* wine\_data\_df = spark.createDataFrame(RDD, schema=["col1", "col2"])
* wine\_data\_df\_pandas = wine\_data\_df.toPandas()
* cluster\_centers\_pandas = pd.DataFrame(model.clusterCenters, columns=["col1", "col2"])
* cluster\_centers\_pandas.head()
* plt.scatter(wine\_data\_df\_pandas["col1"], wine\_data\_df\_pandas["col2"];
* plt.scatter(cluster\_centers\_pandas["col1"], cluster\_centers\_pandas["col2"], color="red", marker="x");

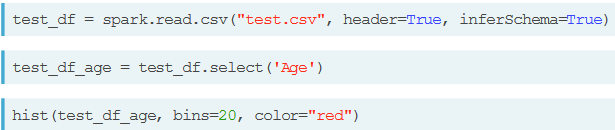
### 5000 points

* 
* 
* 

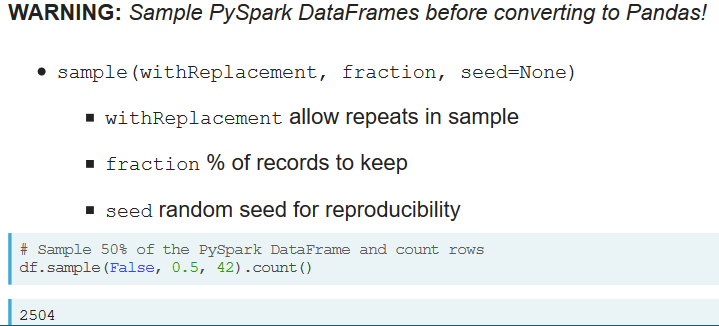
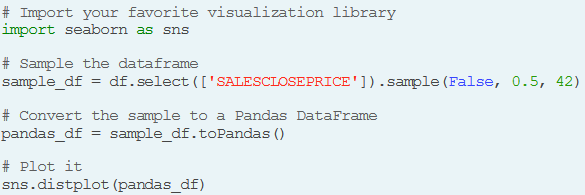
# Data Visualization

* Plotting graphs using PySpark DataFrames is done using three methods
  + pyspark\_dist\_explore library
  + toPandas()
  + HandySpark library

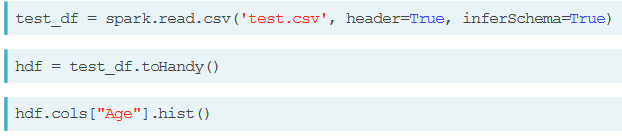
## pyspark\_dist\_explore

* Currently three functions available – hist(), distplot() and pandas\_histogram()
* 

## ToPandas

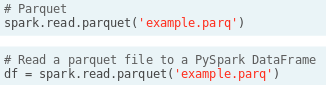
* 
* 

## HandySpark

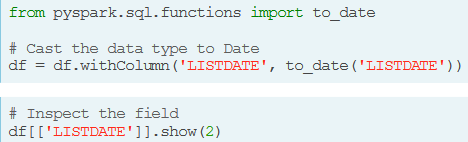
* 

# Feature engineering

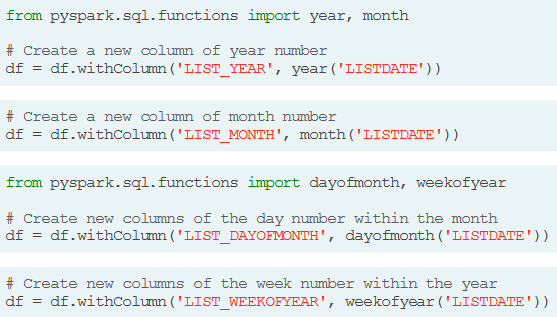
## Parquet

* Write once, read many times
* Stored column-wise
  + Fast to query column subsets, unlike csv
* Structured, defined schema
  + Fields and Data Types defined, saving users from having to define data-types like dates, booleans or strings
  + However, that makes Parquet slow to write
* Not delimited by characters
  + Less likely to be read in wrong if those characters exist in the data
  + Therefore, great for messy text data
* 

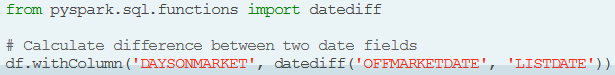
## Date

* 

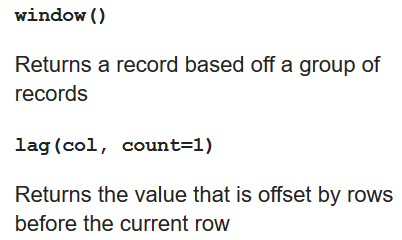
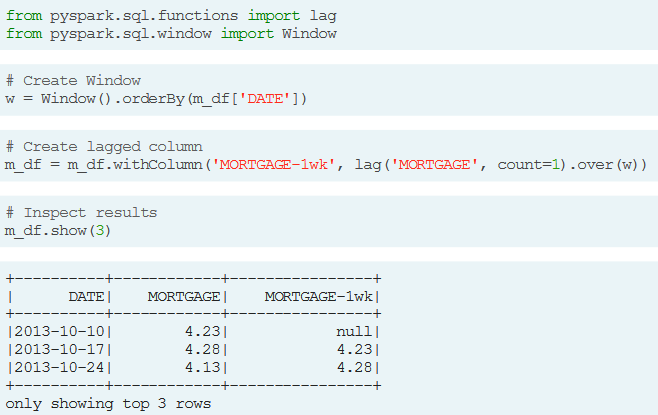
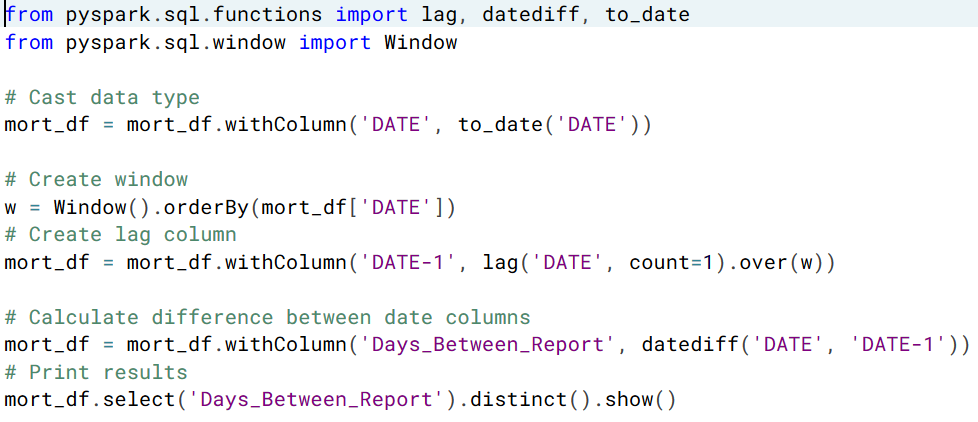
### Time components

* 

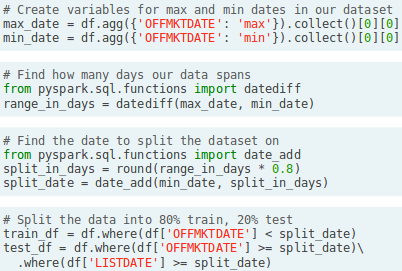
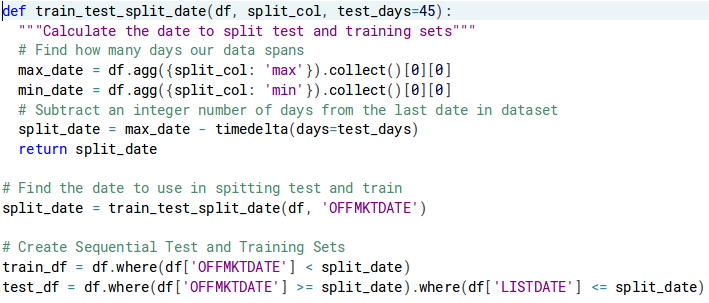
### Diff

* 

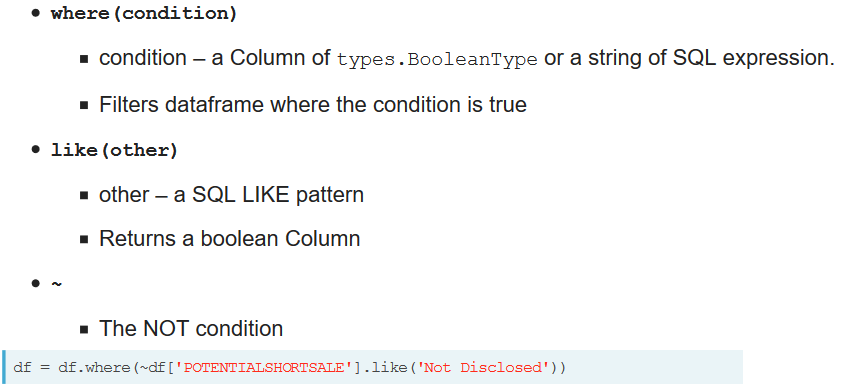
### Lag

* 
* 
* 

### Split

* 
* 

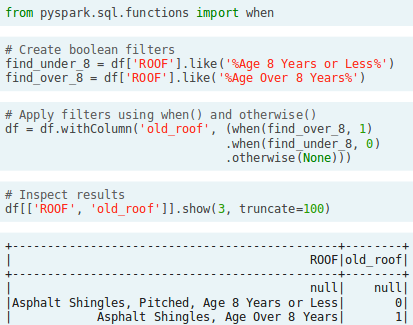
## Text filtering

* 

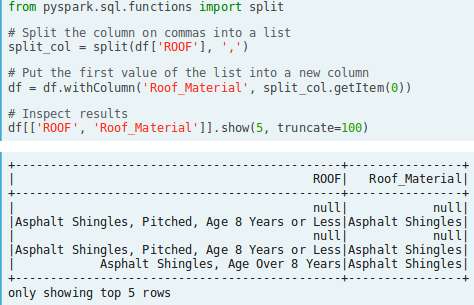
### Like

* 
  + The “%” above is an wildcard, indicating any number of characters before or after the string

### When

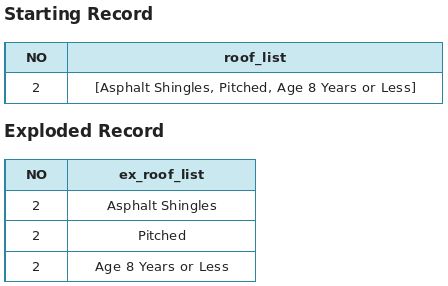
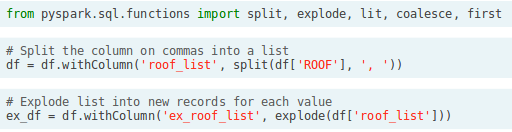
* 

### Split

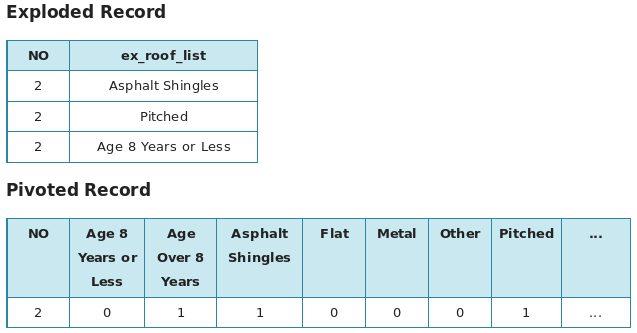
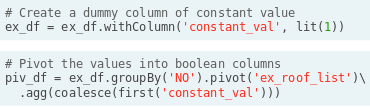
* 

### Explode & Pivot

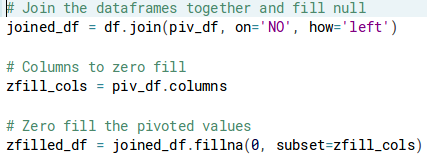
#### Explode

* 
* 
* 

#### Pivot

* 
* 
  + lit() function is used to allow single values where an entire column is expected in a function call

#### Join

* 

## Value filtering

* 