# Introduction to PySpark (Loading Data and PySpark Introduction)

## Creating a Spark Session

We've already created a `SparkSession` for you called `spark`, but what if you're not sure there already is one? Creating multiple `SparkSession`s and `SparkContext`s can cause issues, so it's best practice to use the `SparkSession.builder.getOrCreate()` method. This returns an existing `SparkSession` if there's already one in the environment, or creates a new one if necessary!

```
# Import SparkSession from pyspark.sql
from pyspark.sql import SparkSession

# Create my_spark
my_spark = SparkSession.builder.getOrCreate()

# Print my_spark
print(my_spark)

# Print the tables in the catalog
print(spark.catalog.listTables())
```

One of the advantages of the DataFrame interface is that you can run SQL queries on the tables in your Spark cluster. If you don't have any experience with SQL, don't worry, we'll provide you with queries!

As you saw in the last exercise, one of the tables in your cluster is the `flights` table. This table contains a row for every flight that left Portland International Airport (PDX) or Seattle-Tacoma International Airport (SEA) in 2014 and 2015.

Running a query on this table is as easy as using the `.sql()` method on your `SparkSession`. This method takes a string containing the query and returns a DataFrame with the results!

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.

Remember, we've already created a `SparkSession` called `spark` in your workspace. (It's no longer called `my_spark` because we created it for you!)

```
# Don't change this query
query = "FROM flights SELECT * LIMIT 10"

# Get the first 10 rows of flights
flights10 = spark.sql(query)

# Show the results
flights10.show()
```

## Pandafy a spark DataFrame

Suppose you've run a query on your huge dataset and aggregated it down to something a little more manageable.

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. It's as simple as that!

This time the query counts the number of flights to each airport from SEA and PDX.

Remember, there's already a `SparkSession` called `spark` in your workspace!

```
# Don't change this query
query = "SELECT origin, dest, COUNT(*) as N FROM flights GROUP BY origin, dest"

# Run the query
flight_counts = spark.sql(query)

# Convert the results to a pandas DataFrame
pd_counts = flight_counts.toPandas()

# Print the head of pd_counts
print(pd_counts.head())
```

## Put some Spark in your Data

In the last exercise, you saw how to move data from Spark to `pandas`. However, maybe you want to go the other direction, and put a `pandas` DataFrame into a Spark cluster! The `SparkSession` class has a method for this as well.

The `.createDataFrame()` method takes a `pandas` DataFrame and returns a Spark DataFrame.

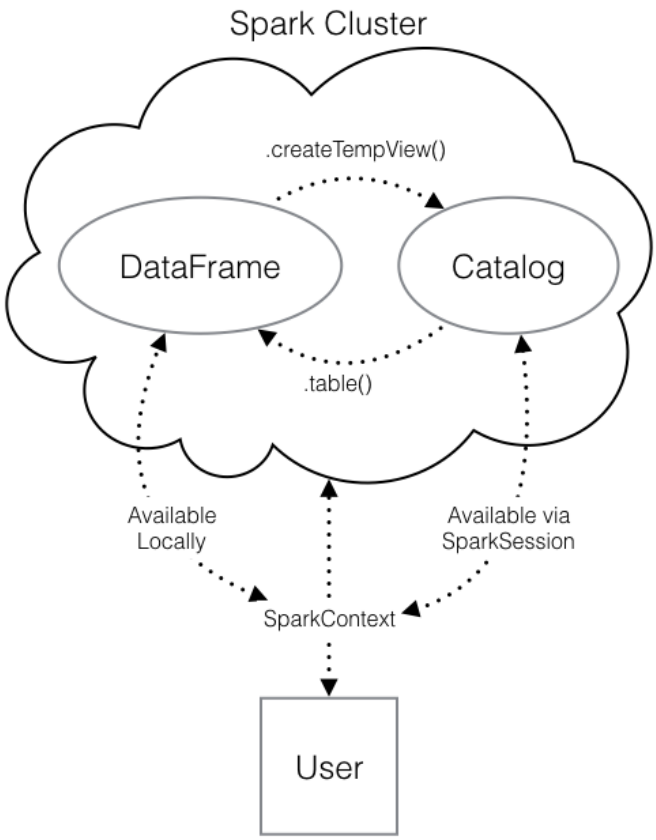
The output of this method is stored locally, not in the `SparkSession` catalog. This means that you can use all the Spark DataFrame methods on it, but you can't access the data in other contexts.

For example, a SQL query (using the `.sql()` method) that references your DataFrame will throw an error. To access the data in this way, 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.

Check out the diagram to see all the different ways your Spark data structures interact with each other.




There's already a `SparkSession` called `spark` in your workspace, `numpy` has been imported as `np`, and `pandas` as `pd`.

```
# Create pd_temp
pd_temp = pd.DataFrame(np.random.random(10))

# Create spark_temp from pd_temp
spark_temp = spark.createDataFrame(pd_temp)

# Examine the tables in the catalog
print(spark.catalog.listTables())

# Add spark_temp to the catalog
spark_temp.createOrReplaceTempView('temp')

# Examine the tables in the catalog again
print(spark.catalog.listTables())
```

## Read DataFrame into Spark directly

Now you know how to put data into Spark via `pandas`, but you're probably wondering why deal with `pandas` at all? Wouldn't it be easier to just read a text file straight into Spark? Of course it would!

Luckily, your `SparkSession` has a `.read` attribute which has several methods for reading different data sources into Spark DataFrames. Using these you can create a DataFrame from a .csv file just like with regular `pandas` DataFrames!

The variable `file_path` is a string with the path to the file `airports.csv`. This file contains information about different airports all over the world.

A `SparkSession` named `spark` is available in your workspace.

```
# Don't change this file path
file_path = "/usr/local/share/datasets/airports.csv"

# Read in the airports data
airports = spark.read.csv(file_path, header=True)

# Show the data
airports.show()
```