Practical 5

In last practical, we tried the Spark RDD. In this practical, you are going to use another important Spark data structure – DataFrame. We'll use Spark to perform some basic analytics a Kaggle dataset called the World Happiness Report. If you download from Kaggle, note that we'll be using the **world-happiness-report.csv** file.

There are two ways you can load the data into a Spark Dataframe. The first one is to use RDD as the medium, this method works regardless of the Spark version.

First, copy the csv file into the **/home/prac/prac5/input** directory. Then, load the csv data into Spark RDD named WHData.

```
>>> WHData = sc.textFile("file:///home/prac/prac5/input/world-happiness-
report.csv").map(lambda x: x.split(","))
```

Then, you can use .toDF() command to convert your RDD into a Dataframe by providing a schema.

```
>>> df = WHData.toDF(["CountryName", "Year", "LifeLadder", "GDP",
"SocialSupport", "LifeExpect", "Freedom", "Generosity", "Corruption",
"Positive", "Negative"])
```

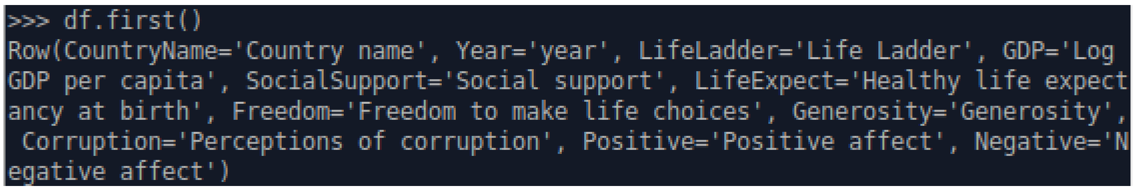
It's a good idea when loading a dataFrame to make sure that all data is the correct format. Take a look now at the type of all columns in the dataframe using:

```
>>> df.printSchema()
```

We can do this with the **cast** command. The command to change the type of the column is below, and to save you from extra work I can tell you that the only column we use this practical that needs re-formatting is the Life Ladder column.

```
>>> df = df.withColumn("LifeLadder", df["LifeLadder"].cast("double"))
```

If you pay a closer look to the DataFrame, you will find the first row of data is actually now actually data. It is the header in the data file.



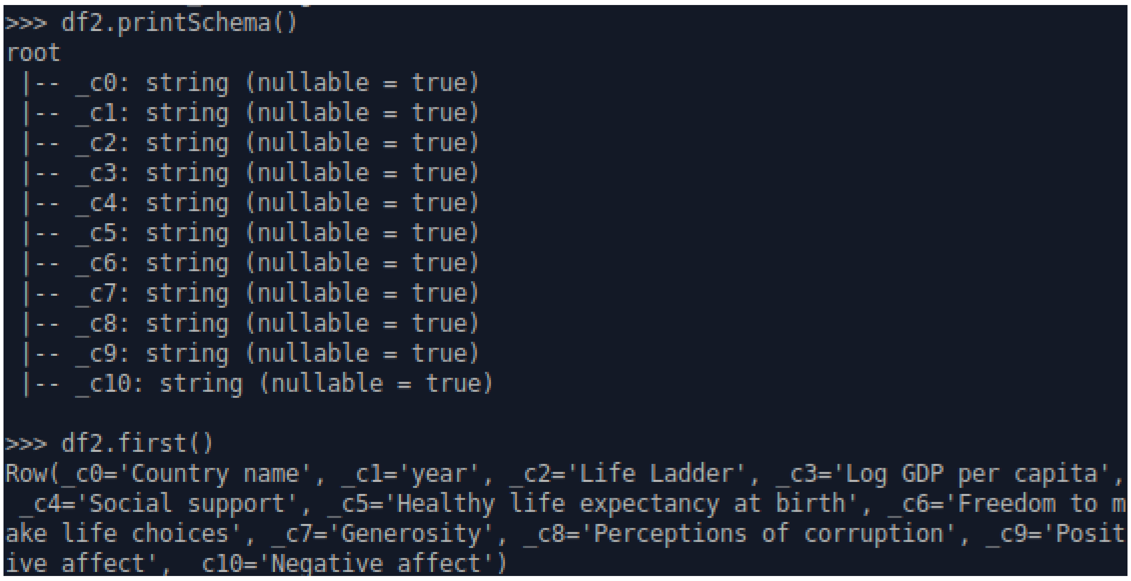
```
>>> df.first()
Row(CountryName='Country name', Year='year', LifeLadder='Life Ladder', GDP='Log
GDP per capita', SocialSupport='Social support', LifeExpect='Healthy life expect
ancy at birth', Freedom='Freedom to make life choices', Generosity='Generosity',
 Corruption='Perceptions of corruption', Positive='Positive affect', Negative='N
egative affect')
```

Therefore, for the first method to work, you need to remove the first line in your data file before loading it into your Spark RDD.

Let’s talk about the second method. The second method uses the **csv()** command in the **pyspark.sql.DataFrameReader** class. You can get access to the csv() command using the **spark** session.

```
>>> df2 = spark.read.csv("file:///home/prac/prac5/input/world-happiness-report.csv")
```

By default, the **csv()** command will not parse the csv header from the file. You can print out the schema and look at the first row of data.



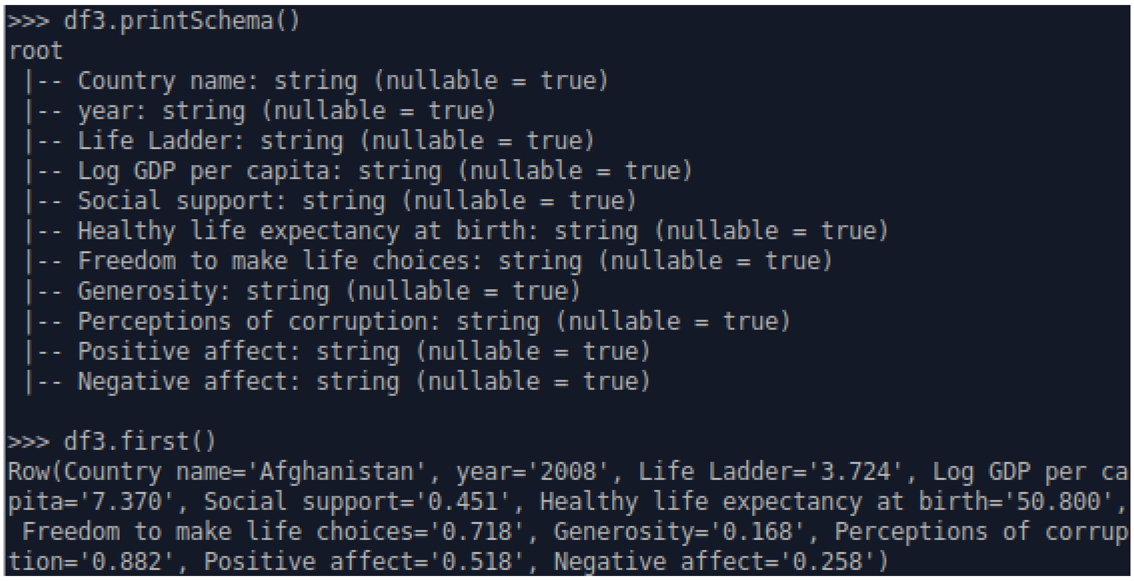
```
>>> df2.printSchema()
root
 |-- _c0: string (nullable = true)
 |-- _c1: string (nullable = true)
 |-- _c2: string (nullable = true)
 |-- _c3: string (nullable = true)
 |-- _c4: string (nullable = true)
 |-- _c5: string (nullable = true)
 |-- _c6: string (nullable = true)
 |-- _c7: string (nullable = true)
 |-- _c8: string (nullable = true)
 |-- _c9: string (nullable = true)
 |-- _c10: string (nullable = true)

>>> df2.first()
Row(_c0='Country name', _c1='year', _c2='Life Ladder', _c3='Log GDP per capita',
 _c4='Social support', _c5='Healthy life expectancy at birth', _c6='Freedom to m
ake life choices', _c7='Generosity', _c8='Perceptions of corruption', _c9='Posit
ive affect', _c10='Negative affect')
```

The **csv()** command reads the data into DataFrame columns "_c0" for the first column and "_c1" for the second and so on. And by default data type for all these columns is treated as String.

You can change the default behaviour by adding an option to the **csv()** command using the **option()** command.

```
>>> df3 = spark.read.option("header", True).csv("file:///home/prac/prac5/input/world-happiness-report.csv")
```

Now your csv file with header is properly parsed into a Spark DataFrame.



```
>>> df3.printSchema()
root
 |-- Country name: string (nullable = true)
 |-- year: string (nullable = true)
 |-- Life Ladder: string (nullable = true)
 |-- Log GDP per capita: string (nullable = true)
 |-- Social support: string (nullable = true)
 |-- Healthy life expectancy at birth: string (nullable = true)
 |-- Freedom to make life choices: string (nullable = true)
 |-- Generosity: string (nullable = true)
 |-- Perceptions of corruption: string (nullable = true)
 |-- Positive affect: string (nullable = true)
 |-- Negative affect: string (nullable = true)

>>> df3.first()
Row(Country name='Afghanistan', year='2008', Life Ladder='3.724', Log GDP per ca
pita='7.370', Social support='0.451', Healthy life expectancy at birth='50.800',
 Freedom to make life choices='0.718', Generosity='0.168', Perceptions of corrup
tion='0.882', Positive affect='0.518', Negative affect='0.258')
```

Now let's try some basic tasks with the data. I'll walk through how my thought process would go when tackling such problems.

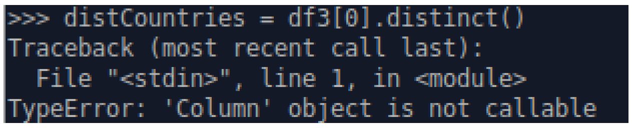
1. Determine how many countries are in the dataset

Surely there's some transformation for getting the unique elements from an RDD? In python we'd use something like **set**. Looking through the list of transformations in the databricks pdf I find the **distinct** transformation, which returns only the distinct elements of an RDD.

In this case we don't want to run the whole dataFrame, just the first column. So we could try something like this:

```
>>> distCountries = df3[0].distinct()
```

However, we'll be met with an error stating 'Column object is not callable'.



```
>>> distCountries = df3[0].distinct()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Column' object is not callable
```

This is because the **distinct** transformation requires an RDD as input. We can reference the RDD behind a DataFrame by calling the **.rdd** command. Then we can use **map** transformation to extract the first column.

```
>>> countriesRDD = df3.rdd.map(lambda x: x[0])
```

Verify the result using the **collect** action.

```
>>> countriesRDD.collect()
```

This is a standard way to get data out of our dataFrame and into an RDD, you should get in the habit of doing it regularly. You can check it worked by typing **countriesRDD** into your terminal. We can try the **distinct** transformation again now.

```
>>> distCountries2 = countriesRDD.distinct()
```

And to check if that worked we can try the take action again.

```
>>> distCountries2.take(5)
```

Finally, we can use the simple count action learned previously to get the number of unique countries in the dataset.

```
>>> distCountries2.count()
```

2. Count how many times each country appears in the dataset

This looks like a problem we're used to, a key-value pair, where the key is the country and the value is a 1 when it appears, then sum the values for each key. First lets map each country to have a one next to it, to get those key-value pairs.

```
>>> countryKVP = df3.rdd.map(lambda x: (x[0],1))
```

Then we use the **reduceByKey** transformation.

```
>>> numPerCountry = countryKVP.reduceByKey(lambda v1, v2: v1+v2)
```

If the **lambda** function in particular here looks a bit tricky, you are encourage to learn a bit more here on the lambda function, which comes very handy when you just need a one-line anonymous function.

https://realpython.com/python-lambda/

Once again, we can use the take action to have a look at some results.

```
>>> numPerCountry.take(5)
```

As an aside, we could have easily performed both transformations at once using a chain operation.

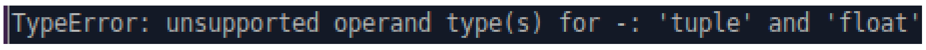
```
>>> numPerCountry = df3.rdd.map(lambda x: (x[0], 1)).reduceByKey(lambda
v1, v2: v1+v2)
```

3. Find the mean number of times a country appears, then create a new RDD that adds the mean to each count

Looking at the documentation for the **mean** action, we can see that it takes the mean of the whole RDD. Trying it on our result from the last question.

>>> numPerCountry.mean()

This will yield an error stating ‘unsupported operand for -: 'tuple' and 'float'’.



```
TypeError: unsupported operand type(s) for -: 'tuple' and 'float'
```

Instead we need to create an RDD containing just the values. This is straight forward using the map transformation.

```
>>> justValues = numPerCountry.map(lambda x: x[1])
```

```
>>> justValues.mean()
```

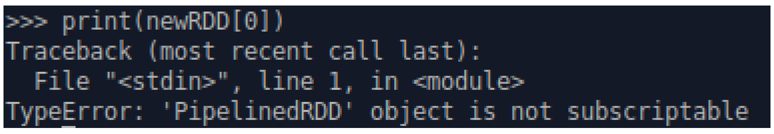
Since we want to use this value and not just look at it, we can assign it to a variable.

```
>>> M = justValues.mean()
```

It seems straightforward to add this value to the previous ones, lets use map to print the country name, then add M to the country count of our previous RDD.

```
>>> newRDD = numPerCountry.map(lambda x: (x[0], x[1]+M))
```

Finally, what if we wanted to print some of these values alongside text using Python? Or manipulate them while printing? We’ll get an error stating that ‘PipelinedRDD’ object is not support subscriptable.



```
>>> print(newRDD[0])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'PipelinedRDD' object is not subscriptable
```

We can’t pass an RDD into a **print** command, or do any elementwise operations on an RDD. Instead we’d have to collect the values, then the list elements can be accessed and manipulated as we normally would using square bracket notation.

```
>>> myList = newRDD.collect()
```

```
>>> print("Hello " + str(myList[0][0]))
```

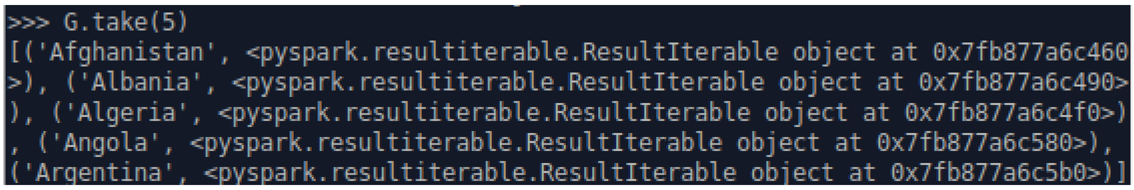
4. Find which country has the most life ladder values

This could be done fairly easily using previous results, but lets try another approach. This sounds like we could group the data so that there's a list of values for each country. The **groupByKey** transformation sounds appropriate. Let's make a set of key-value pairs for country name and life ladder, then group that RDD by key.

```
>>> ladderKVP = df3.rdd.map(lambda x: (x[0], x[2]))
```

```
>>> G = ladderKVP.groupByKey()
```

This seems to work fine, but when we take a closer look at G using **G.take(5)** we see that alongside each country is not a list of numbers but an iterable object.



```
>>> G.take(5)
[('Afghanistan', <pyspark.resultiterable.ResultIterable object at 0x7fb877a6c460
>), ('Albania', <pyspark.resultiterable.ResultIterable object at 0x7fb877a6c490>
), ('Algeria', <pyspark.resultiterable.ResultIterable object at 0x7fb877a6c4f0>)
, ('Angola', <pyspark.resultiterable.ResultIterable object at 0x7fb877a6c580>),
('Argentina', <pyspark.resultiterable.ResultIterable object at 0x7fb877a6c5b0>)]
```

In order to access their values for manipulation or aggregation we need to convert that object to a list.

```
>>> G2 = G.map(lambda x: (x[0], list(x[1])))
```

Try taking the top 5 elements again and we should see something that aligns better with our expectations.

Next we want to get the length of the values in the list for each country.

```
>>> G3 = G2.map(lambda x: (x[0], len(x[1])))
```

Finally, we can use the max action to get the largest value.

```
>>> G3.max()
```

The result is Zimbabwe,15 but we already know that Canada also appears 15 times. It seems like max is finding the maximum letter value in the alphabet of the competing options.

We can give the **max** action an optional function to indicate what data should be used to find the maximum value.

```
>>> G3.max(key=lambda x: x[1])
```

Useful Resources

https://www.kaggle.com/

https://training.databricks.com/visualapi.pdf

https://spark.apache.org/docs/latest/api/python/reference/index.html

https://realpython.com/python-lambda/