Lab : Apache Spark Math and String Functions

Pre-reqs:

- Google Chrome (Recommended)

Prerequisites

We need following packages to perform the lab exercise:

- Java Development Kit
- pyspark

JAVA

Verify the installation with: `java -version`

You'll see the following output:

```
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)
```

Math Functions

There are a number of math functions which can be applied to columns with numbers. Let us now look at few of them.

Fire up the spark-shell from the terminal `spark-shell`

Step 1: Let us first create a collection with data as shown below. Please make sure you have the imports from the previous section already imported. You will have to import them again if you have closed the Spark Session.

`val numbers = List(5, 4, 9.4, 25, 8, 7.7, 6, 52)`

Step 2: Next, let us convert the collection to dataset using the toDS method and rename the column as numbers using the withColumnRenamed method. The default column name when you create a dataset is value. Hence we change the default column name to numbers.

`val numbersDS = numbers.toDS().withColumnRenamed("value", "numbers").cache()`

The dataset should now be created with the renamed column.

Step 3: Let us now perform various math functions on the dataset. All these functions are self explanatory.

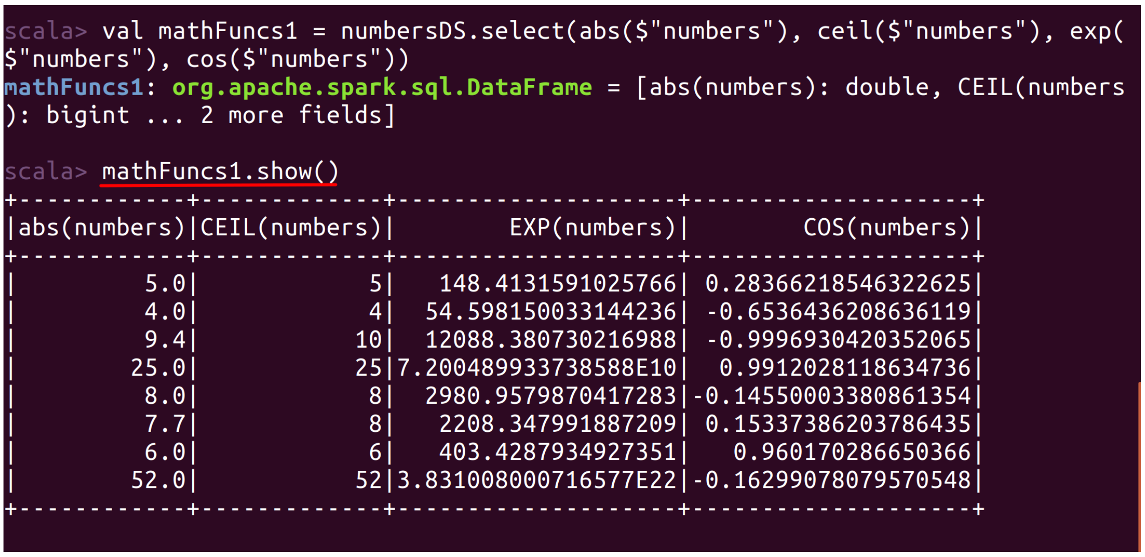
`val mathFuncs1 = numbersDS.select(abs($"numbers"), ceil($"numbers"), exp($"numbers"), cos($"numbers"))`

- The abs function returns the absolute value of the number.
- The ceil function returns the number of double type greater than or equal to the nearest rounded integer.
- The exp function returns Eulers E raised to power of double value.
- The cos function returns the trigonometric cosine of an angle.

Let us check the result using the show method.

```
mathFuncs1.show()
```

The following result is shown.



```
scala> val mathFuncs1 = numbersDS.select(abs($"numbers"), ceil($"numbers"), exp(
$"numbers"), cos($"numbers"))
mathFuncs1: org.apache.spark.sql.DataFrame = [abs(numbers): double, CEIL(numbers
): bigint ... 2 more fields]

scala> mathFuncs1.show()
+------------+-------------+-------------------+--------------------+
|abs(numbers)|CEIL(numbers)|        EXP(numbers)|         COS(numbers)|
+------------+-------------+-------------------+--------------------+
|         5.0|            5|   148.4131591025766| 0.28366218546322625|
|         4.0|            4|   54.598150033144236| -0.6536436208636119|
|         9.4|           10|   12088.380730216988| -0.9996930420352065|
|        25.0|           25|7.200489933738588E10|   0.9912028118634736|
|         8.0|            8|   2980.9579870417283|-0.14550003380861354|
|         7.7|            8|    2208.347991887209| 0.15337386203786435|
|         6.0|            6|    403.4287934927351|    0.960170286650366|
|        52.0|           52|3.831008000716577E22|-0.16299078079570548|
+------------+-------------+-------------------+--------------------+
```

Step 4: Let us now use some more math functions.

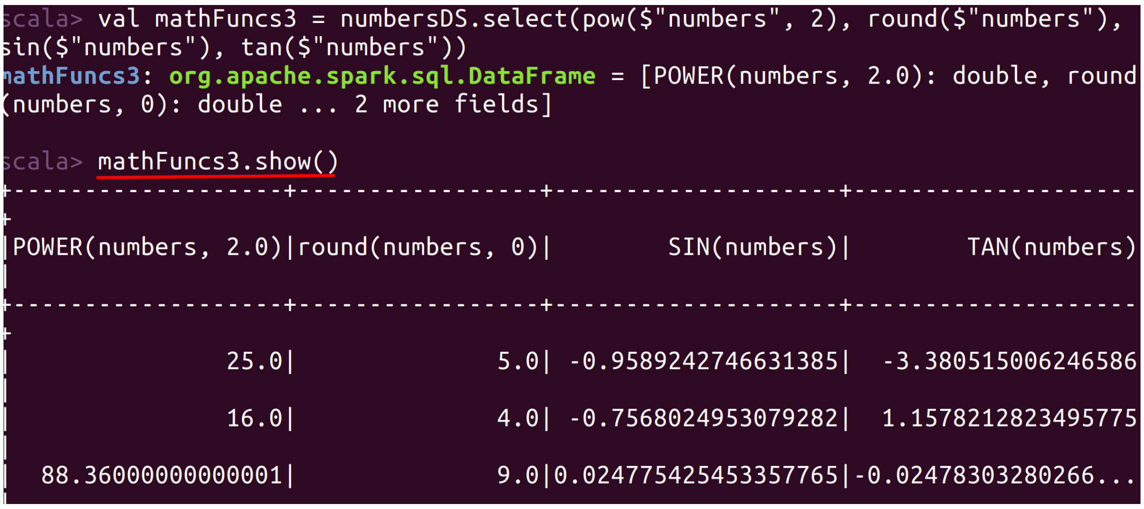
```
val mathFuncs2 = numbersDS.select(factorial($"numbers"), floor($"numbers"),
hex($"numbers"), log($"numbers"))
```

- The factorial functions returns the factorial of the number.
- The floor function is opposite to the ceil function which returns the number of double type lesser than or equal to the nearest rounded integer.
- The hex function returns a hex value.
- The log function returns the natural logarithm (base e) of a double value as a parameter.

Let us check the result using the show method.

```
mathFuncs2.show()
```

The following result is shown.



```
scala> val mathFuncs3 = numbersDS.select(pow($"numbers", 2), round($"numbers"),
sin($"numbers"), tan($"numbers"))
mathFuncs3: org.apache.spark.sql.DataFrame = [POWER(numbers, 2.0): double, round
(numbers, 0): double ... 2 more fields]

scala> mathFuncs3.show()
+-------------------+-----------------+-------------------+-------------------
+
|POWER(numbers, 2.0)|round(numbers, 0)|       SIN(numbers)|       TAN(numbers)
|
+-------------------+-----------------+-------------------+-------------------
+
|               25.0|              5.0| -0.9589242746631385| -3.380515006246586
|
|               16.0|              4.0| -0.7568024953079282|  1.1578212823495775
|
|  88.36000000000001|              9.0|0.024775425453357765|-0.02478303280266...
```

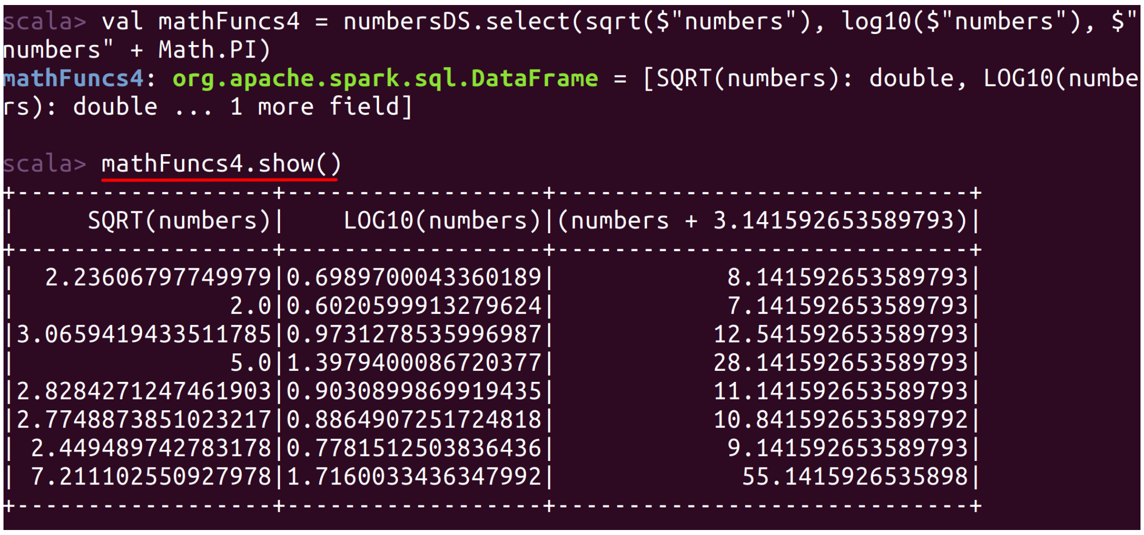
Step 5: Let us now use even more math functions.

```
val mathFuncs3 = numbersDS.select(pow($"numbers", 2), round($"numbers"),
sin($"numbers"), tan($"numbers"))
```

- The pow function returns the number raised to the power of some other number. It takes two arguments. The first argument is the column with numbers and the second argument is number which the power has to be calculated.
- The round function returns the rounded value to its nearest decimal.
- The sin and tan functions return the sine and tangent trignometric angle respectively.

Let us check the result using the show method. `mathFuncs3.show()`

The following result is shown.



```
scala> val mathFuncs4 = numbersDS.select(sqrt($"numbers"), log10($"numbers"), $"
numbers" + Math.PI)
mathFuncs4: org.apache.spark.sql.DataFrame = [SQRT(numbers): double, LOG10(numbe
rs): double ... 1 more field]

scala> mathFuncs4.show()
+------------------+------------------+----------------------------+
|     SQRT(numbers)|    LOG10(numbers)|(numbers + 3.141592653589793)|
+------------------+------------------+----------------------------+
|  2.23606797749979|0.6989700043360189|           8.141592653589793|
|               2.0|0.6020599913279624|           7.141592653589793|
|3.0659419433511785|0.9731278535996987|          12.541592653589793|
|               5.0|1.3979400086720377|          28.141592653589793|
|2.8284271247461903|0.9030899869919435|          11.141592653589793|
|2.7748873851023217|0.8864907251724818|          10.841592653589792|
| 2.449489742783178|0.7781512503836436|           9.141592653589793|
| 7.211102550927978|1.7160033436347992|            55.1415926535898|
+------------------+------------------+----------------------------+
```

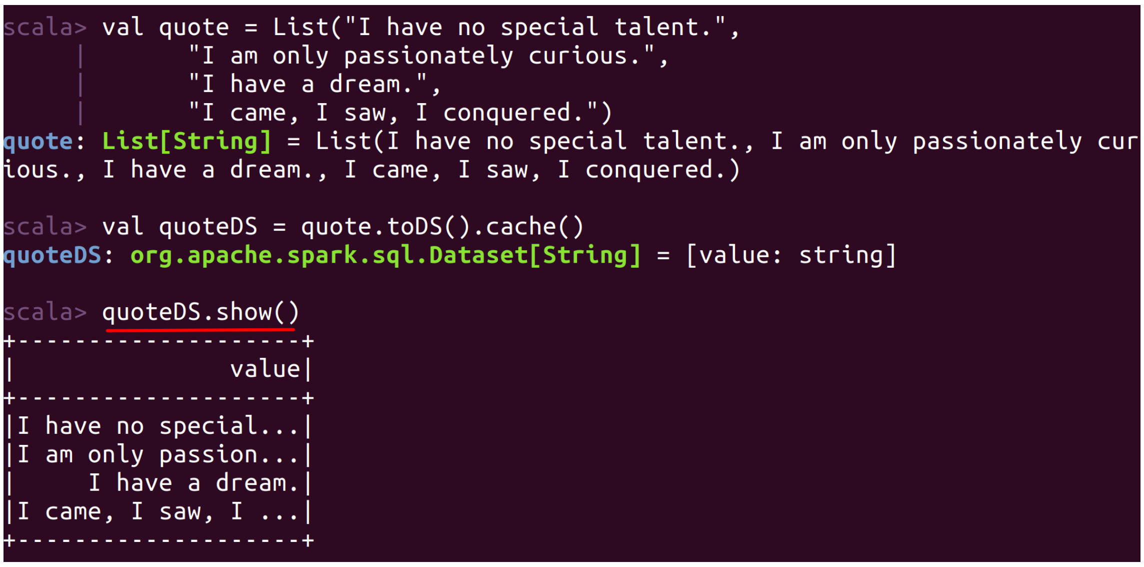
Step 6: Let us finally conclude math functions with a couple more of them.

`val mathFuncs4 = numbersDS.select(sqrt($"numbers"), log10($"numbers"), $"numbers" + Math.PI)`

- The sqrt function returns the square root of the given numbers in the column.
- The log10 function returns the base 10 logarithm of a double value.
- In the third column, we have simply added the value of PI by using the Math.PI expression.

Let us check the result using the show method. `mathFuncs4.show()`

The following result is shown.



```
scala> val quote = List("I have no special talent.",
     |        "I am only passionately curious.",
     |        "I have a dream.",
     |        "I came, I saw, I conquered.")
quote: List[String] = List(I have no special talent., I am only passionately cur
ious., I have a dream., I came, I saw, I conquered.)

scala> val quoteDS = quote.toDS().cache()
quoteDS: org.apache.spark.sql.Dataset[String] = [value: string]

scala> quoteDS.show()
+--------------------+
|               value|
+--------------------+
|I have no special...|
|I am only passion...|
|     I have a dream.|
|I came, I saw, I ...|
+--------------------+
```

String Functions

There are a plethora of String functions available in Spark. Let us look at few of them now.

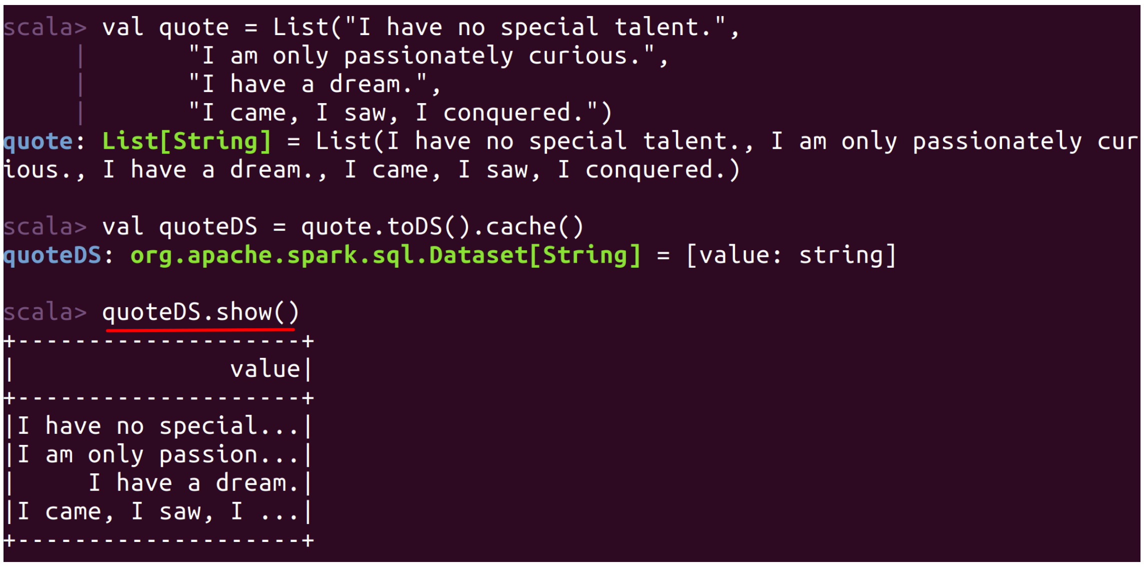
Step 1: As usual, let us first create the List and create a dataset from it. Please make sure to specify imports again if you have closed the Spark session.

```
val  quote = List("I have no special talent.",
  "I am only passionately curious.",
  "I have a dream.",
  "I came, I saw, I conquered.")
```

`val quoteDS = quote.toDS().cache()`

Let us use the show method to display the dataset as shown below.

`quoteDS.show()`



```
scala> val quote = List("I have no special talent.",
     |        "I am only passionately curious.",
     |        "I have a dream.",
     |        "I came, I saw, I conquered.")
quote: List[String] = List(I have no special talent., I am only passionately cur
ious., I have a dream., I came, I saw, I conquered.)

scala> val quoteDS = quote.toDS().cache()
quoteDS: org.apache.spark.sql.Dataset[String] = [value: string]

scala> quoteDS.show()
+--------------------+
|               value|
+--------------------+
|I have no special...|
|I am only passion...|
|     I have a dream.|
|I came, I saw, I ...|
+--------------------+
```

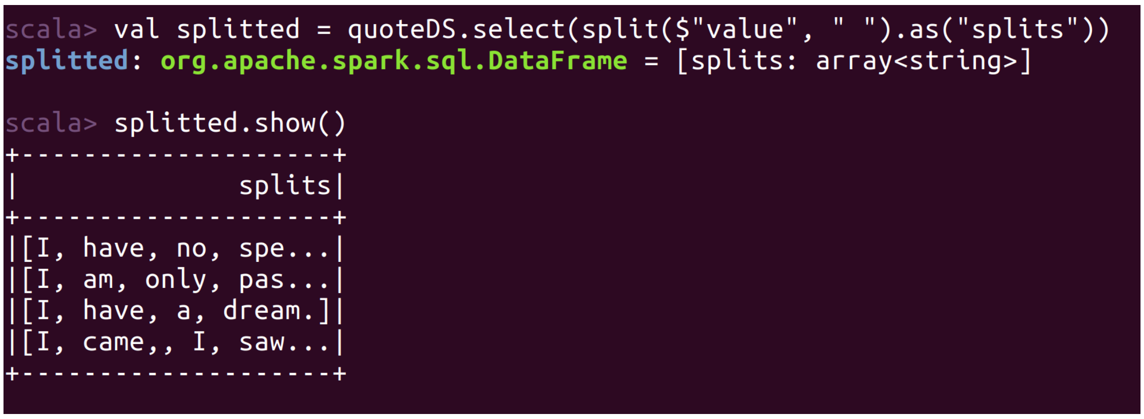
Step 2: Let us first use the split method to split the strings by space as below.

```
val splitted = quoteDS.select(split($"value", " ").as("splits"))
```

The split method takes two arguments. The first argument is the name of the column and the second method is the pattern to which the string should be split on.

Let us use the show method to display the dataset as shown below.

```
splitted.show()
```



```
scala> val splitted = quoteDS.select(split($"value", " ").as("splits"))
splitted: org.apache.spark.sql.DataFrame = [splits: array<string>]

scala> splitted.show()
+--------------------+
|              splits|
+--------------------+
|[I, have, no, spe...|
|[I, am, only, pas...|
|[I, have, a, dream.]|
|[I, came,, I, saw...|
+--------------------+
```

As you can see from the screenshot above, the rows have now been splitted by whitespace.

Step 3: Next, let us use the explode function which we have learned in this exercise to create column for each element in the collection.

```
val exploded = splitted.select(explode($"splits").as("explode"))
```

Let us use the show method to display the dataset as shown below.

```
exploded.show()
```

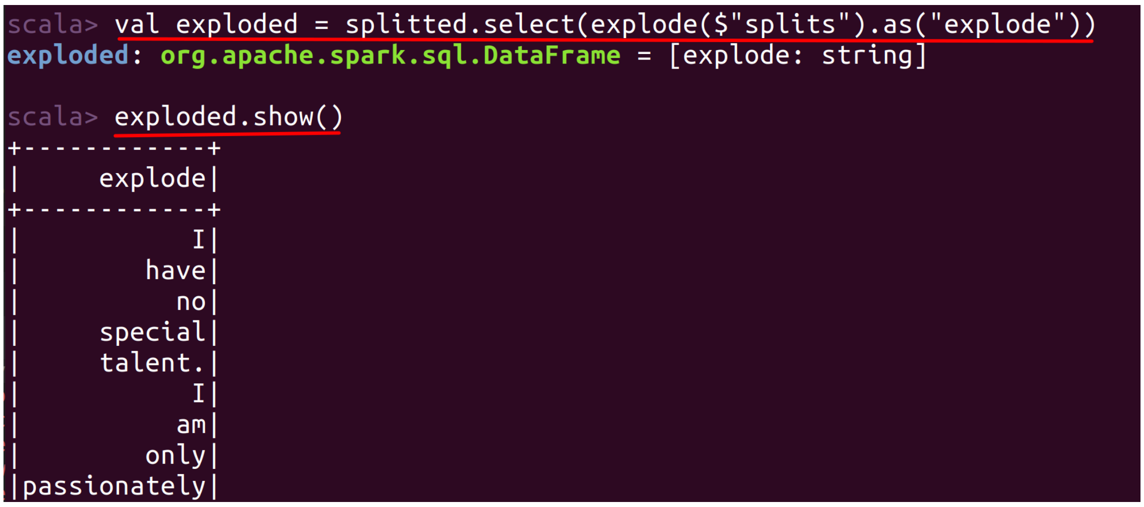
Now that we have each word as a row, let us apply some string functions.

The first function we use is to find out the length of each word using the length function as shown below.

```
val strLen = exploded.select($"explode", length($"explode")).as("length")
```

Let us use the show method to display the dataset as shown below.

```
strLen.show()
```



```
scala> val exploded = splitted.select(explode($"splits").as("explode"))
exploded: org.apache.spark.sql.DataFrame = [explode: string]

scala> exploded.show()
+------------+
|     explode|
+------------+
|           I|
|        have|
|          no|
|     special|
|     talent.|
|           I|
|          am|
|        only|
|passionately|
```

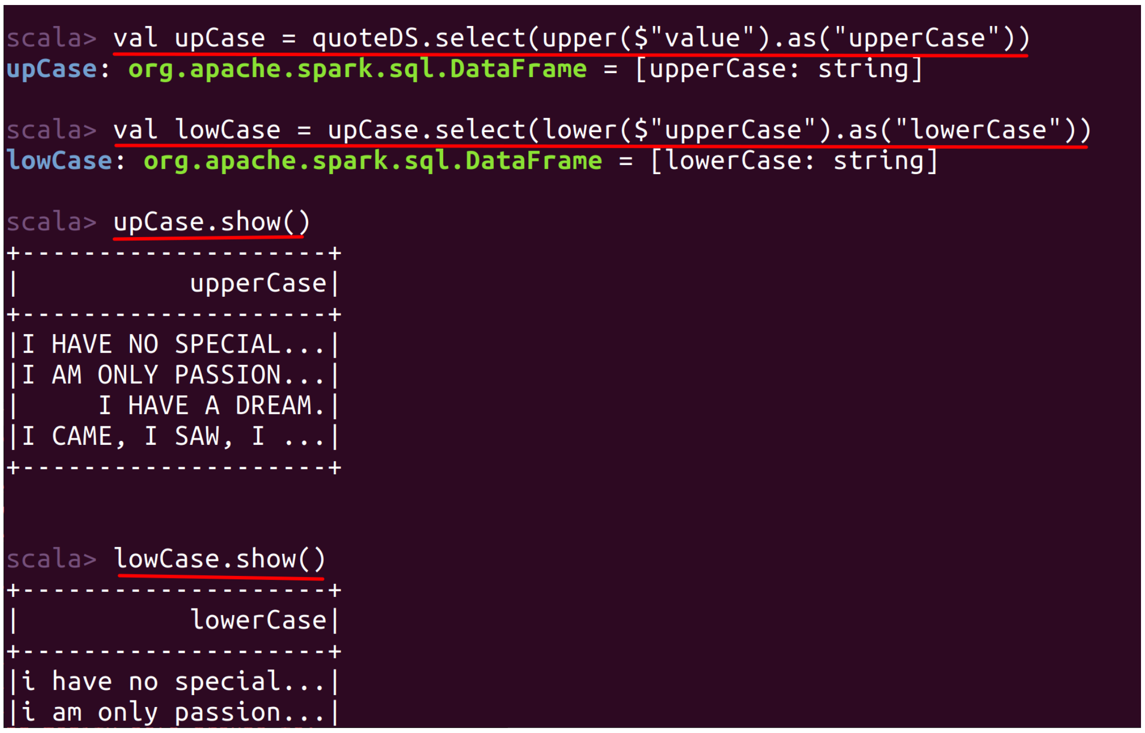
Step 4: Let us now use string functions to convert strings from lower case and upper case.

```
val upCase = quoteDS.select(upper($"value").as("upperCase"))
```

```
val lowCase = upCase.select(lower($"upperCase").as("lowerCase"))
```

Let us use the show method to display the datasets as shown below.

```
upCase.show()
lowCase.show()
```



```
scala> val upCase = quoteDS.select(upper($"value").as("upperCase"))
upCase: org.apache.spark.sql.DataFrame = [upperCase: string]

scala> val lowCase = upCase.select(lower($"upperCase").as("lowerCase"))
lowCase: org.apache.spark.sql.DataFrame = [lowerCase: string]

scala> upCase.show()
+--------------------+
|           upperCase|
+--------------------+
|I HAVE NO SPECIAL...|
|I AM ONLY PASSION...|
|     I HAVE A DREAM.|
|I CAME, I SAW, I ...|
+--------------------+


scala> lowCase.show()
+--------------------+
|           lowerCase|
+--------------------+
|i have no special...|
|i am only passion...|
```

Step 5: Finally, let us look at substring and trim functons to extract a part of string and trim the whitespaces before and after the string respectively.

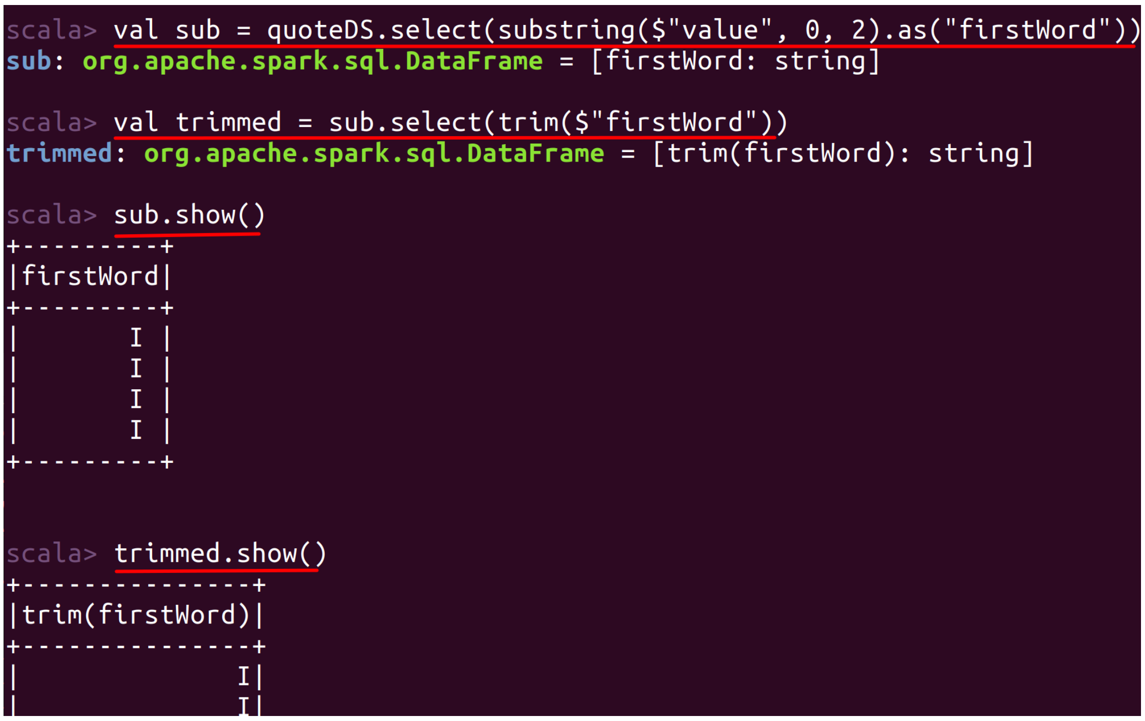
```
val sub = quoteDS.select(substring($"value", 0, 2).as("firstWord"))
```

```
val trimmed = sub.select(trim($"firstWord"))
```

Let us use the show method to display the datasets as shown below.

```
sub.show()
trimmed.show()
```

The substring function takes three arguments. The first is the column name from which the sub string to be extracted from. Second and third are the start and positions from which we want to extract the string from.



```
scala> val sub = quoteDS.select(substring($"value", 0, 2).as("firstWord"))
sub: org.apache.spark.sql.DataFrame = [firstWord: string]

scala> val trimmed = sub.select(trim($"firstWord"))
trimmed: org.apache.spark.sql.DataFrame = [trim(firstWord): string]

scala> sub.show()
+---------+
|firstWord|
+---------+
|       I |
|       I |
|       I |
|       I |
+---------+


scala> trimmed.show()
+---------------+
|trim(firstWord)|
+---------------+
|              I|
|              I|
```

Here we have extracted the first word and space after it using the substring function. Since our fist word is only a letter, we start from 0 position and end at 2nd position which is the whitespace. Next, we use the trim function to trime the whitespaces before and after. Since there are no whitespaces before, the function will simply trim the whitespace after. You can also use rtrim function to trim only the whitespaces at the end and ltrim function to trim only the whitespaces at the beginning.

Task is complete!