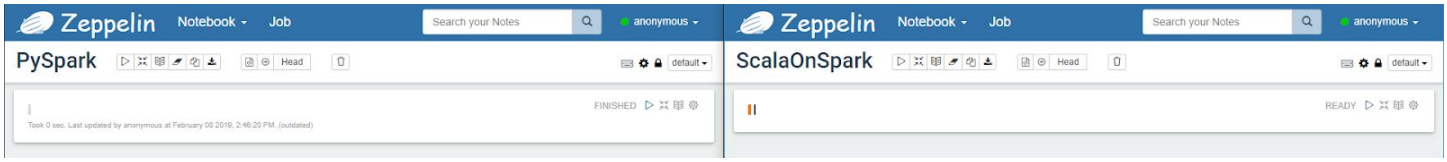
Spark Introduction

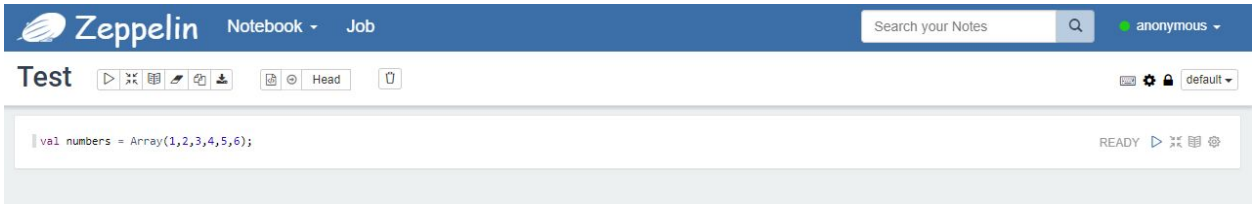
DS730

In this activity, you will be running some Spark code on the local Hortonworks system. We will be interacting with Spark using Apache Zeppelin.

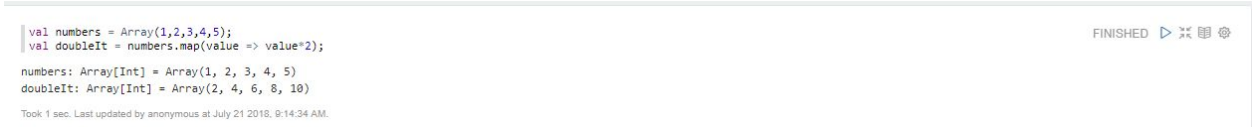
1. Apache Spark is written in Scala. This step will show you how to run Scala code on Spark and also how to run Python code on Spark (i.e. PySpark). Go to http://localhost:9995 and you will be at the Zeppelin UI. On the homepage, click on **Create new note**. You can name your note anything you want but just be sure that **spark2** is the default interpreter. You are encouraged to create a second note in a second window using spark2 as the interpreter for that note as well. That way you can run Scala code and Python code at the same time:



   For those of you who have used Jupyter or some other notebook, this UI will be very familiar to you. If you want to run 1 command at a time, you will enter in 1 command and press the “play” button on the right hand side to the right of READY:



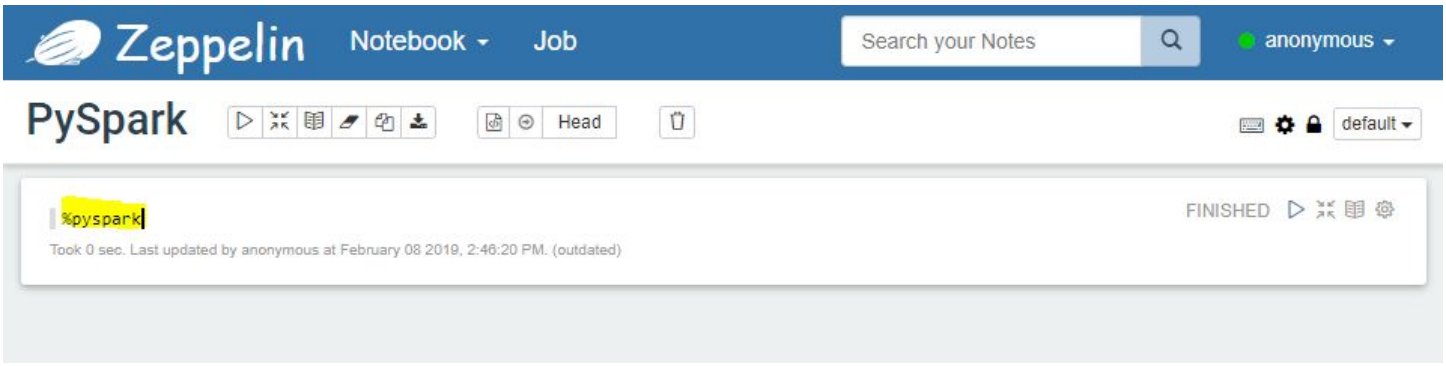
   The action will be pending and eventually execute depending on the complexity of the commands. You can put in multiple steps and run them:



   We will spend quite a bit of time going through the syntax now. Learning syntax can be a bit dry but it is a necessary step we have to take. Feel free to pose questions on the discussion board if you are curious about how a particular statement works or can work differently.

   Regular Python code can be written in a PySpark environment. In order to see something new, much of the Python code in this activity will contain lambda expressions. Such expressions are quite common in PySpark code. If you have not seen them before, their usage is explained later in this activity.

The activity will be formatted in the following fashion. A step will contain a Scala statement in green. The subsequent step will contain an identical Python statement in blue. The subheading that follows will explain what is going on with the statements. You can run these one at a time or put in multiple commands to see them running as a group. There will be simple explanations of some functions to start but more detail on these functions will appear as you go through the activity. The only small change to start is that in your PySpark note, be sure that %pyspark is the first line of your program:



To get a rundown of all of the functions available to you, be sure to check out the guides located at: https://spark.apache.org/docs/latest/quick-start.html

You can approach these steps however you like. You can either run through all of the green Scala on Spark code and repeat and do the blue PySpark code. Or you can do both of them at the same time to see the difference in syntax.

When running the Scala code, it will automatically display the contents of each line. PySpark does not do this. You need to add print statements to each thing you want to print out. For example, for step b, you would need to write print(numbers) if you wanted to see what numbers contained.

a. **val numbers = Array(1,2,3,4,5,6)**
    i. A simple array to see how some functions work. You notice numbers type is an Array[Int]. It is very important that you know exactly what types you are working with or certain functions will not work as expected.

b. **numbers = [1,2,3,4,5,6]**
    i. A simple list in Python.

c. **val doubleIt = numbers.map(value => value * 2)**

i. The numbers.map part is saying that we want to take all of the values in the numbers array and copy them to a new array. However, we don't want to simply copy the values in the array to the new array. The **map** function works on each value in the numbers array and tells the mapper how to copy the values. The syntax may be a bit confusing. Nothing is particularly special about the value => value*2 syntax. The map function expects a function to be passed in. The value => value*2 is itself a function. You are telling the mapper exactly how to map each value from the original array to the new array. The map function is saying: for every value in the array, double it. Then store the newly formed array into a variable called doubleIt.

d. **doubleIt = map(lambda value: value*2, numbers)**
   i. If you are not familiar with lambda functions, the two lines of Python code in part (b) and (d) can be translated to the Python code below:

```
numbers = [1,2,3,4,5,6]
doubleIt = []
for value in numbers:
  doubleIt.append(value*2)
```

e. **val small = doubleIt.filter(value => value < 5)**
   i. A simple **filter** function that does what it sounds like it will do. It filters out all values greater than or equal to 5. It only keeps values that are less than 5. The result is stored into a variable called small.

f. **small = filter(lambda value: value<5, doubleIt)**
   i. Same as the previous step.

g. **val moreNumbers = doubleIt.union(numbers)**
   i. The union method does what you would expect: it creates the union of both lists. However, note that these are not sets. There is a 2 in each list. Therefore, two 2's show up in moreNumbers. If you do not want both 2's, use the following: **val moreNumbers = doubleIt.union(numbers).distinct**

h. **moreNumbers = list(set(doubleIt).union(numbers))**
   i. Same as previous step with distinct.

i. **val total = doubleIt.reduce((a,b) => a + b)**
   i. The reduce function, similar to map, takes a function as an argument. However, the reduce function will "reduce" the array down to 1 value. The way reduce

works is that it takes the first 2 elements in the array and calls them ‘a’ and ‘b.’ Reduce will act on those values. In this case, it will add them together. That result will become the new ‘a’ value. Reduce will take that newly created ‘a’ value and look at the third element as ‘b.’ It will add them together to create a new ‘a’ value and this continues until the entire array has been iterated over.

j. **reduce(lambda a,b: a+b, moreNumbers)**

i. Reduce is another function that uses lambda functions[1]. A comparable Python program would look like this:

```
numbers = [1,2,3,4,5,6]
a = numbers[0] + numbers[1]
index = 2
while index < len(numbers):
  b = numbers[index]
  a = a + b
  index += 1
print(a)
```

k. **val wordsFile = sc.textFile("/user/maria_dev/wcinput/wap.txt")**

i. Our first basic command is to load up a file using the Spark context, denoted **sc**. sc is a built-in variable that has a lot of nice functions available for us to use. The one we are using here is **textFile**. We pass in the file location and the file is "stored" in the wordsFile variable. Assuming all went well, you will see something like this:

wordsFile: org.apache.spark.rdd.RDD[String] = /user/maria_dev/wcinput/wap.txt MapPartitionsRDD[1] at textFile at <console>:27

ii. The object that was returned is something called an RDD (resilient distributed dataset). RDDs are created by reading in files from our local filesystem or from HDFS or from transforming other RDDs. **RDDs are immutable**. An RDD is not an array but that is not a bad way to think about it. An RDD is a partitioned collection of elements. In this case, it is a collection of Strings. Each line is one of the partitions.
Once an RDD is created, it cannot be changed. However, one can transform an RDD to create a new RDD. For example, we could filter out all lines that contained the word "cat." In doing so, our original RDD is unchanged but a newly

[1] reduce has been moved to a module since Python3. In order to use it with regular Python3, you must import functools and call the reduce function in that module. For example: *import functools as ft* and then *ft.reduce(...)*.

created RDD is created and that new RDD will only contain rows with the word cat in them. We'll see an example of that in a few steps.

l. **wordsFile = sc.textFile("/user/maria_dev/wcinput/wap.txt")**
   i. Same as previous step.

m. **wordsFile.count()**
   i. The count function tells you how many lines were in the file. You should have 66055.

n. **wordsFile.count()**
   i. Same as previous step.

o. **wordsFile.collect()**
   i. This function will display the contents of the RDD. If there are too many lines to show, the display will end with … as it does in this case. Each element in the array is a line in the original text file.

p. **wordsFile.collect()**
   i. Same as previous step. Because we are printing out a ton of data here, it is possible that PySpark will not display it. It may simply display a warning. This is fine and you can move to the next step.

q. **val onlyWar = wordsFile.filter(line => line.contains("war"))**
   i. The first part of the command says to filter so as you can imagine, we are filtering the rows (or lines) of the file. The filter function creates a new RDD such that the new RDD contains filtered lines. The original RDD is unchanged. This is where it gets a bit strange so please read the following carefully.
   Think back to step (k), the textFile function expected 1 argument. That argument was a file location. You had to tell the textFile function where the file was located in order for the textFile function to work. In a similar fashion, the filter function needs 1 argument. The argument you pass into a filter function is itself a function. More on that function in a minute.
   The RDD that was created in step (k) was an RDD[String]. In other words, the elements of the RDD are Strings (i.e. the rows of the text file). Since our RDD is an RDD[String], the function that you pass in to the filter function has the form of (String ⇒ Boolean). In other words, a String (in our case, a line of the input) is passed in and a Boolean value is returned. The filter function then takes that true/false value and uses it to determine if the line belongs in the new RDD. If the value is true, it puts it in the new RDD.

The function has the syntax of (X ⇒ Boolean). In our case, we just called the line of text *line*. There is nothing special about the word line, the function could have just as easily been:
(xyz => xyz.contains("war"))
The righthand side is fairly straightforward then: line.contains("war") returns true if *war* is in the line, false otherwise. The filter function then uses that information to only include lines that contained *war*. Be careful with contains, if the line contained “down*war*d” then contains would be true.
The concept of passing functions into functions is not a trivial one. These functions are called *higher-order functions*. If the map, reduce and filter functions do not make sense, you are encouraged to post your questions on the discussion board. Simply doing a search for higher-order functions can lead you down a rabbit hole that gets weird in a hurry if it’s not explained well enough.

r. **onlyWar = wordsFile.filter(lambda line: "war" in line)**
   i. Same as previous step.

s. **onlyWar.count()**
   i. This will print out the number of rows that contained the word war. In our case, it was 1219.

t. **onlyWar.count()**
   i. Same as previous step.

u. **wordsFile.filter(line => line.contains("war")).count()**
   i. We can also chain commands together. Instead of creating a (potentially) useless variable, we can print out the count of the newly created RDD and throw away the temporary RDD immediately.

v. **wordsFile.filter(lambda line: "war" in line).count()**
   i. Same as previous step.

w. **val tempWord = "This is a test"**
   i. A simple example to create a String.

x. **tempWord = "This is a test"**
   i. Same as previous step.

y. **val tempWordArray = tempWord.split(" ")**

i. Another example that shows how split works. An Array[String] is the type of tempWordArray.

z. **tempWordArray = tempWord.split(" ")**
   i. Same as previous step.

aa. **val numWords = tempWordArray.size**
   i. A simple example showing how size counts the number of strings in the tempWordArray array. These simple concepts are used together the next Scala example.

bb. **len(tempWordArray)**
   i. Same as previous step.

cc. **val mappedSize = wordsFile.map(line => line.split(" ").size)**
   i. A map is similar to a filter. It is a function that expects a function as an argument. A map differs in that the function that is passed in is of the form (X ⇒ Y) where X is the type of the original RDD and Y is anything you want it to be. In our case, we have (String ⇒ Integer). Our mapping function accepts a string and returns the number of words in that string. line.split(" ") will return an array of Strings and the size function will count the number of words in that array. Therefore, every row in our input file will be mapped to the number of Strings that appear in that row.

dd. **mappedSize = wordsFile.map(lambda line: len(line.split()))**
   i. Same as previous step.

ee. **val reduced = mappedSize.reduce((first,second) => if(first > second) first else second)**
   i. The reduce function is similar to the map and filter function. The reduce function is generally used to aggregate the elements in some fashion. It accepts a function as an argument. The function is of the form: ((X, X) ⇒ X). In our example, mapped is an RDD[Integer]. The function looks at 2 integers in the RDD and returns the larger of the two. In our example, it compares row 1 and row 2 and gets the larger of the two values. Let's call that value m. Then it compares m to row 3 and takes the largest value and so on. Assuming you ran the code that was in step (f), can you come up with 1 line of code that produces the same answer as this step. In other words, can you create 1 line of code that combines step (o) and step (p). The answer to this question is at the end of the activity.

ff. **mappedSize.reduce(lambda first, second: first if (first > second) else second)**
    i. Same as previous step.

gg. **val flattenMap = wordsFile.flatMap(line => line.split(" "))**
val mapped = wordsFile.map(line => line.split(" "))
    i. As with other languages we have learned, it is very important to know the types of data you are working with. The difference between an RDD[String] and an RDD[Array[String]] is very important. An RDD[String] is an RDD that contains a String for each word in the file. An RDD[Array[String]] is an RDD that contains an Array of Strings. This is the difference between map and flatMap. In this example, when you use map, you get back an RDD[Array[String]]. In other words, every "row" in the RDD is an array and that array is a list of strings. For each line in the original input file, map will take it, split the lines by spaces and return an array of strings for that row. If you run: mapped.count() you will end up with 66055 which is the **number of lines** in the file. Whereas flattenMap is an RDD[String]. This means that the array of strings that map gives you is flattened to produce 1 big array of strings. Instead of an array for each line, flatMap will take all of the strings and put them into 1 large array. Your output from flattenMap.count() is 583077 which is the **number of words** in the entire file.

hh. **flattenMap = wordsFile.flatMap(lambda line: line.split())**
mapped = wordsFile.map(lambda line: line.split())
    i. Same as previous step.

ii. **val mapFirst = flattenMap.map(word => (word, 1))**
    i. A simple example to see how the mapping in MapReduce works. This is the map step.

jj. **mapFirst = flattenMap.map(lambda word: (word, 1))**
    i. Same as previous step.

kk. **val reduceSecond = mapFirst.reduceByKey((a,b) => a+b)**
    i. A simple example to see how the reducing in MapReduce works.

ll. **reduceSecond = mapFirst.reduceByKey(lambda x, y: x+y)**
    i. Same as previous step.

mm. **val mapreduce = flattenMap.map(word => (word, 1)).reduceByKey((a,b) => a+b)**

i. We have seen map and reduce separately in steps (ii) and (kk) but this step is where they are combined. The MapReduce framework works well with Spark. In this example, we will take our RDD[String] and map every word to a (key, value) pair which is just the word as the key followed by a 1. Instead of storing that into a temporary RDD we are immediately acting on it by calling the reduceByKey function. The reduceByKey function accepts a function per usual. The function accepts 2 values (i.e. (a,b) in this example) and does something to them. In our case, we simply add them together. This differs from reduce in that we are grouping together the keys first and then reducing the values of each specific key. reduceByKey assumes the first value in the tuple is the key and the second value is the value. For example, if the key "cat" appears 5 times in our file, reduceByKey will take ("cat",1), ("cat",1), ("cat",1), ("cat",1), ("cat",1) and produce ("cat", 5). You can view the contents of the variable by doing a mapreduce.collect(). Because there are hundreds of thousands of words, this command might take a minute.

nn. **mapreduce = flattenMap.map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b)**
mapreduce.collect()
    i. Combines steps (jj) and (ll) similar to the previous step.

2. This section will show you how to use **Datasets** in Spark using Scala. As of now, Python does not support Datasets. For this portion, we will be using New York City taxi data to try and answer some questions about taxi rides in NYC.
    a. Download the dataset into your **HDFS** to location **/user/maria_dev/taxi.csv** but note that this is a large dataset (1.6GB):
    https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2016-06.csv

    b. Before we load our data, we need to define our Dataset. We are essentially defining the columns of our data and the types of those columns:
    case class TaxiRides(VendorID: Integer,
    tpep_pickup_datetime: String,
    tpep_dropoff_datetime: String,
    passenger_count: Integer,
    trip_distance: Double,
    pickup_longitude: Double,
    pickup_latitude: Double,
    RatecodeID: Integer,
    store_and_fwd_flag: String,
    dropoff_longitude: Double,
    dropoff_latitude: Double,
    payment_type: Integer,

```
fare_amount: Double,
extra: Double,
mta_tax: Double,
tip_amount: Double,
tolls_amount: Double,
improvement_surcharge: Double,
total_amount: Double)
```

c. We need to import a library so that we can use a Dataset:
import org.apache.spark.sql._

d. We want to load our data into Spark using the following command:
val taxi_ds: Dataset[TaxiRides] = spark.sqlContext.read.option("header", "true").option("delimiter", ",").option("inferSchema", "true").csv("/user/maria_dev/taxi.csv").as[TaxiRides]

There are three options we are specifying. The first option is the **header** which says our csv file has a header row. The second option tells our reader what character is **delimiting** the columns. The third option tells Spark to **inferSchema** which is self-explanatory. The **csv** tells Spark what csv file we want to open and finally the **as[TaxiRides]** tells Spark which Dataset to use. Since this file is quite large, it will take a couple of minutes to load.

e. There are many functions that we can apply to the Dataset. Since we are dealing with a larger dataset and we are running this in a virtual machine, these commands may take a few minutes to execute. Several of the common Dataset functions are shown below:
   i. **taxi_ds.show(50)**
   Shows the first 50 rows.

   ii. **taxi_ds.count()**
   Display the number of rows in the file. There should be 11135470 rows.

   iii. **taxi_ds.filter(x => x.tip_amount>=300).show()**
   Filter the data to only include rows that have a tip_amount greater than or equal to 300. When that data is calculated, show it. There are 11 such rows.

   iv. **taxi_ds.map(x => x.total_amount).reduce((x,y) => x+y)**
   As of this writing, reduce is still an experimental function so it may not operate as expected. You can find details on reduce (and the other functions available to Datasets) here:
   https://spark.apache.org/docs/latest/api/Scala/index.html#org.apache.spark.sql.Dataset
   The functions are simple map and reduce functions that we saw earlier in the activity. The map function takes a row and turns it into a single value (the total amount). The reduce function takes all of those total amount values and sums them up. This gives us

the total amount made by all taxi drivers that month. The number I got was 1.8741174523132128E8, which is roughly $187 million.

v. **taxi_ds.agg(sum("total_amount")).show()**
Another nice function is the **agg** function that allows us to do some nice aggregations of the data. For example, the previous mapreduce problem can be solve with an agg function. This example gives the same result (as expected) as the previous mapreduce call.

vi. **taxi_ds.groupBy("VendorID").agg(sum("total_amount")).show()**
This last function shows how you can group your data together and then run some aggregate function on it. In this example, there are only 2 vendor id's, 1 and 2. They are grouped together and the total amount for each vendor is summed. The results are: (1, 8.792306356965834E7), (2, 9.948868165957363E7)

f. There are several other functions that one can use with Datasets. The link above provides documentation of the ones you are able to use. The majority of ones you want to use have been demonstrated in this activity.

3. This section will show you how to run SQL queries using Spark using Scala and Python. For this portion, we will again be using New York City taxi data to try and answer some questions about taxi rides in NYC. We are using a **DataFrame** in this example. A DataFrame is similar to a table in a normal RDBMS. DataFrames have row objects and a datatype that is associated with each column. Because the code is almost identical for Scala and Python, the commands will be shown in the same step with the green code being Scala code and blue code being Python code.
   a. We want to load our data into Spark using the following command:
   val taxi = spark.read.format("csv").option("header", true).option("inferSchema", true).load("/user/maria_dev/taxi.csv")
   taxi = spark.read.format("csv").option("header", "true").option("inferSchema", "true").load("/user/maria_dev/taxi.csv")

   We are reading in our data that is in a csv file. There are two options we are specifying. The first one is the **header** option which says our csv file has a header row. The second option tells Spark to **inferSchema** which is self-explanatory. If you do not infer the schema or tell Spark what a schema is, it will assume all of your columns are strings. This is problematic when it comes to integers/doubles. The load tells Spark where our file is located.

   b. If we want to view the schema of our data, we run the following command:
   taxi.printSchema()
   taxi.printSchema()

   c. Our DataFrame object can be acted upon like it were a regular database. Therefore, one can select, group by, filter, etc. The following are examples of how to use each common function. In

Scala, you reference the columns using the $"ColName" syntax and you use nameOfDataFrame["ColName"] syntax in Python.

i. **taxi.select($"passenger_count", $"tip_amount").show()**
taxi.select(taxi["passenger_count"], taxi["tip_amount"]).show()

This command shows all of the values in the *passenger_count* and *tip_amount* column. Thankfully, the Zeppelin notebook will only show the first 20 rows. You can add as many columns to that command as you want. Just separate them by commas.

ii. **taxi.filter($"passenger_count" > 5).show()**
taxi.filter(taxi["passenger_count"] > 5).show()

This command will include the rows where the passenger_count was greater than 5. You can chain these commands together to produce something like this:

taxi.filter($"passenger_count" > 5).filter($"tip_amount" > 10).select($"passenger_count", $"tip_amount").show()
taxi.filter(taxi["passenger_count"] > 5).filter(taxi["tip_amount"] > 10).select(taxi["passenger_count"], taxi["tip_amount"]).show()

iii. A final thing that you might want to do with your DataFrame is group the data by some column. Let's say you drive for a ride sharing company and you want to know which fare to take based on how many passengers there are. Maybe a certain number of passengers tips better. You can group your data by passenger count and then take the average tip that each count gives. Note the $ variable in the Scala code in the groupBy function. However, the $ does not appear in the avg function (the same in Python).

taxi.groupBy($"passenger_count").avg("tip_amount").show()
taxi.groupBy(taxi["passenger_count"]).avg("tip_amount").show()

d. Our taxi object is a DataFrame object and SQL cannot be run directly on a DataFrame object. Therefore, we must create a table or a view to run our queries on:
taxi.createOrReplaceTempView("taxiView")

e. Now that we have a view of our data, we can run any standard SQL queries that we want on it. For example, let's say we want to get the average tip amount based on the passenger count like we did in the previous steps. We use the following command:
spark.sqlContext.sql("SELECT passenger_count, avg(tip_amount) FROM taxiView GROUP BY passenger_count").show()
spark.sql("SELECT passenger_count, avg(tip_amount) FROM taxiView GROUP BY passenger_count").show()

The **spark.sqlContext.sql** is just the start of our command in order to run a query (or spark.sql).

The command itself is a standard SQL query between the double quotes. The SQL query is executed and shown.

f. There are many questions you could ask of the taxi data. For example, what percentage of taxi trips had a toll involved? If you want to check your solution against mine, my solution was roughly 5.66%.

4. Let's say you start working for a ride-sharing company and you want to drive taxi part time in NYC. You would like to maximize the fares that you can get. But you only have 1 hour to work each day. You run some code on your dataset to determine which hour of the day is best to be working. It is relatively easy to come up with code to take the average fare for hours that start and end on the hour (e.g. 2:00pm - 3:00pm). You simply group together all fares that start at hour 2pm and compute the average of the fare. It is less obvious how to write code to calculate the average from 11:17am - 12:17pm. It is even less obvious how to do so for all 60 minute intervals.
A feature we will now look at is something called Windowing. It lets you assign a "window" where you only look at certain rows in a dataset. For instance, only look at rows that contain times that are within 60 minutes of the current row… and do this for all rows. This step will be done with Scala.
For the sake of speed, I am using a smaller dataset called smaller.csv. You can download it from http://www.uwosh.edu/faculty_staff/krohne/ds730/smaller.csv.

   a. **val taxi = spark.read.format("csv").option("header", true).option("inferSchema", true).load("smaller.csv")**
   val stamp = taxi.withColumn("timestamp", unix_timestamp($"tpep_pickup_datetime", "MM/dd/yyyy HH:mm"))

      We will use the taxi variable from the previous step. Our first goal is going to be to create a unix_timestamp from the pickup date/time. The withColumn function adds a new column to the taxi dataset. The information in that column is the unix_timestamp function.

   b. **val windowSpec = Window.partitionBy("VendorID").orderBy("timestamp").rowsBetween(-4000,4000)**

      Window is a built-in class that will help us do our "windowing." The partitionBy function is optional and allows us to partition our input into chunks. In this case, we are putting all rows with a VendorID of 1 into 1 chunk and all rows with a VendorID of 2 in another chunk. A more advanced partition would partition the data by pick-up location so we know where the best place to pick up passengers is.
      The orderBy function tells the window which column we want to order by. In this case, we are ordering by the timestamp that was created in step (a).
      Lastly, the rowsBetween tells the window how many rows should be in our window. The -4000 says to look back 4000 rows from the current row. The 4000 says to look ahead 4000 rows from

the current row. In other words, 8001 rows will be considered in our window[2]. The number 8000 comes from the fact that, on average, roughly 8000 rides are in 1 hour.

c. **val answer = stamp.withColumn("someAverage", avg(stamp("fare_amount")).over(windowSpec))**

This code creates a new aggregated column for us called someAverage. The code will take the average of the fare_amount in the specified window.

d. **answer.select("VendorID", "tpep_pickup_datetime", "timestamp", "fare_amount", "someAverage").show()**

The code above will select certain columns from the answer DataFrame and show them. Your code and output should look something like this:

```
val taxi = spark.read.format("csv").option("header", true).option("inferSchema", true).load("smaller.csv")
val stamp = taxi.withColumn("timestamp", unix_timestamp($"tpep_pickup_datetime", "MM/dd/yyyy HH:mm"))
val windowSpec = Window.partitionBy("VendorID").orderBy("timestamp").rowsBetween(-4000,4000)
val answer = stamp.withColumn("someAverage", avg(stamp("fare_amount")).over(windowSpec))
answer.select("VendorID", "tpep_pickup_datetime", "timestamp", "fare_amount", "someAverage").show()
```

```
taxi: org.apache.spark.sql.DataFrame = [VendorID: int, tpep_pickup_datetime: string ... 17 more fields]
stamp: org.apache.spark.sql.DataFrame = [VendorID: int, tpep_pickup_datetime: string ... 18 more fields]
windowSpec: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@7cfaf86
answer: org.apache.spark.sql.DataFrame = [VendorID: int, tpep_pickup_datetime: string ... 19 more fields]
+--------+--------------------+----------+-----------+------------------+
|VendorID|tpep_pickup_datetime| timestamp|fare_amount|       someAverage|
+--------+--------------------+----------+-----------+------------------+
|       1|       6/1/2016 0:00|1464739200|        7.5|14.002101974506374|
|       1|       6/1/2016 0:00|1464739200|       19.5|14.001476761619191|
|       1|       6/1/2016 0:00|1464739200|       11.5|13.999477891581314|
|       1|       6/1/2016 0:00|1464739200|        5.0|14.003848651348653|
|       1|       6/1/2016 0:00|1464739200|        6.5|14.006968789013733|
|       1|       6/1/2016 0:00|1464739200|        6.5| 14.00521967049426|
|       1|       6/1/2016 0:00|1464739200|        2.5|14.003221861741952|
|       1|       6/1/2016 0:00|1464739200|        9.0|14.002722055888224|
|       1|       6/1/2016 0:00|1464739200|        5.5| 14.00134946370666|
|       1|       6/1/2016 0:00|1464739200|        8.0|14.002970074812968|
|       1|       6/1/2016 0:00|1464739200|        5.0|14.002207678883972|
```

e. **val windowSpec = Window.partitionBy("VendorID").orderBy("timestamp").rangeBetween(-1800,1800)**

In this step, we are going to change the windowSpec so that we aren't blindly looking back 4000 rows and looking ahead 4000 rows. Rather, we are going to use the rangeBetween function. The rangeBetween function says to look back 1800 seconds (or 30 minutes) from the timestamp cell and also look ahead 1800 seconds from the timestamp cell. This is the new window. Instead of guessing that 4000 is the size of 30 minutes worth of data, now we are being explicit and

[2] If there are not 4000 rows before the current row, the window will go all the way back to the first row and stop. In other words, if we are on row 2300, the window would consider rows from 1-2299, 2300 and 2301-6300. It works the same for if there are not 4000 rows after the current row.

saying to go back exactly 30 minutes and go forward exactly 30 minutes. You'll notice the someAverage is identical for all of these rows. Do you understand why?

```
val taxi = spark.read.format("csv").option("header", true).option("inferSchema", true).load("smaller.csv")
val stamp = taxi.withColumn("timestamp", unix_timestamp($"tpep_pickup_datetime", "MM/dd/yyyy HH:mm"))
val windowSpec = Window.partitionBy("VendorID").orderBy("timestamp").rangeBetween(-1800,1800)
val answer = stamp.withColumn("someAverage", avg(stamp("fare_amount")).over(windowSpec))
answer.select("VendorID", "tpep_pickup_datetime", "timestamp", "fare_amount", "someAverage").show()
```

```
taxi: org.apache.spark.sql.DataFrame = [VendorID: int, tpep_pickup_datetime: string ... 17 more fields]
stamp: org.apache.spark.sql.DataFrame = [VendorID: int, tpep_pickup_datetime: string ... 18 more fields]
windowSpec: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.WindowSpec@2d6bcb4f
answer: org.apache.spark.sql.DataFrame = [VendorID: int, tpep_pickup_datetime: string ... 19 more fields]
+--------+--------------------+----------+-----------+------------------+
|VendorID|tpep_pickup_datetime| timestamp|fare_amount|       someAverage|
+--------+--------------------+----------+-----------+------------------+
|       1|       6/1/2016 0:00|1464739200|        7.5|14.204838129496403|
|       1|       6/1/2016 0:00|1464739200|       19.5|14.204838129496403|
|       1|       6/1/2016 0:00|1464739200|       11.5|14.204838129496403|
|       1|       6/1/2016 0:00|1464739200|        5.0|14.204838129496403|
|       1|       6/1/2016 0:00|1464739200|        6.5|14.204838129496403|
|       1|       6/1/2016 0:00|1464739200|        6.5|14.204838129496403|
|       1|       6/1/2016 0:00|1464739200|        2.5|14.204838129496403|
|       1|       6/1/2016 0:00|1464739200|        9.0|14.204838129496403|
|       1|       6/1/2016 0:00|1464739200|        5.5|14.204838129496403|
|       1|       6/1/2016 0:00|1464739200|        8.0|14.204838129496403|
|       1|       6/1/2016 0:00|1464739200|        5.0|14.204838129496403|
```

The last step in our problem is deciding which 60 minute interval is the best one.

5. This step explains some of the other interesting things you can do with Scala on Spark. Many of them have similar Python commands that are shown in blue on the same step. However, many of these should be Python commands that you have already seen.
    a. **wordsFile.map(line => line.length).reduce((first,second) => first+second)**
    wordsFile.map(lambda line: len(line)).reduce(lambda first, second: first+second)
        i. The length function returns the number of letters in the line. The reduce function then adds up the number of characters in the entire file.

    b. **val oneLess = mapreduce.mapValues(value => value-1)**
    oneless = mapreduce.mapValues(lambda value: value-1)
        i. This function allows us to edit all of the values of the mapreduce array without changing the key. A common function that only modifies the value. But remember, the original mapreduce RDD remains unchanged. Only the new oneLess RDD represents this change.

    c. **val allKeys = mapreduce.keys**
    allKeys = mapreduce.keys()
        i. A simple function that returns an Array[String] containing all of the keys from mapreduce. This string contains all of the words in the original text file where each word shows up in the array once (because the keys are unique). There is a similar function called values.

d. **val manyWords = mapreduce.filter{case (key, value) => value > 2000}**
manyWords = mapreduce.filter(lambda (key, value): value > 2000)
    i. Note the syntax is a bit different for this one. We cannot just use the filter function as we did before. Before we had an RDD of strings so it was fine to say filter based on whether or not the string contained the word "war," for example. We need to say we are using a (key,value) pair instead and the syntax on how to do that is above. Once we've specified we are filtering a (key,value) pair, we filter it based on whether or not a word appeared more than 2000 times in the file. As expected, some common words show up in manyWords: (to, he, at, all, by, had...)

e. **manyWords.saveAsTextFile("/user/maria_dev/wap_output/")**
manyWords.saveAsTextFile("/user/maria_dev/wap_output/")
    i. If you want to save your data to a file, you can call the saveAsTextFile function and pass in a directory. As with Hadoop, you must save your data to a folder that has not yet been created. If you try and save it to a folder that already exists, it will not run. You are saving the files to your HDFS filesystem with this function call.

f. **val multFiles = sc.textFile("/user/maria_dev/wap_output/part*")**
multFiles = sc.textFile("/user/maria_dev/wap_output/part*")
    i. You can read in multiple files at once by using wildcards. The above will create 1 RDD with all of the words in that 1 RDD. If you would rather the files be separated, you can use the following:
    val wordsFile = sc.wholeTextFiles("/user/maria_dev/wap_output/part*")
    wordsFile = sc.wholeTextFiles("/user/maria_dev/wap_output/part*")
    This will create an RDD[(String, String)] where the first string is the filename and the second string is the entire file. Note the second value is not an array of strings where each line is 1 array. Rather, the second value is simply a String. When running wordsFile.collect(), this was the output I received:

```
Array((hdfs://sandbox-hdp.hortonworks.com:8020/user/maria_dev/wap_outpu
t/part-00000,"(from,2517)
(as,3694)
(with,5520)
(is,2990)
(him,2722)
(his,7630)
(,16769)
(she,2540)
"),
(hdfs://sandbox-hdp.hortonworks.com:8020/user/maria_dev/wap_output/part
-00001,"(to,16320)
(he,7631)
(at,4201)
(all,2272)
```

```
(the,31704)
(not,4411)
(by,2310)
(it,3325)
(but,2789)
(had,5305)
(said,2406)
(a,10018)
(I,3225)
(The,2550)
(of,14855)
(on,3368)
(you,2417)
(and,20564)
(for,3270)
(in,8228)
(be,2300)
(that,7230)
(was,7188)
(were,2352)
(her,3882)
"))
```

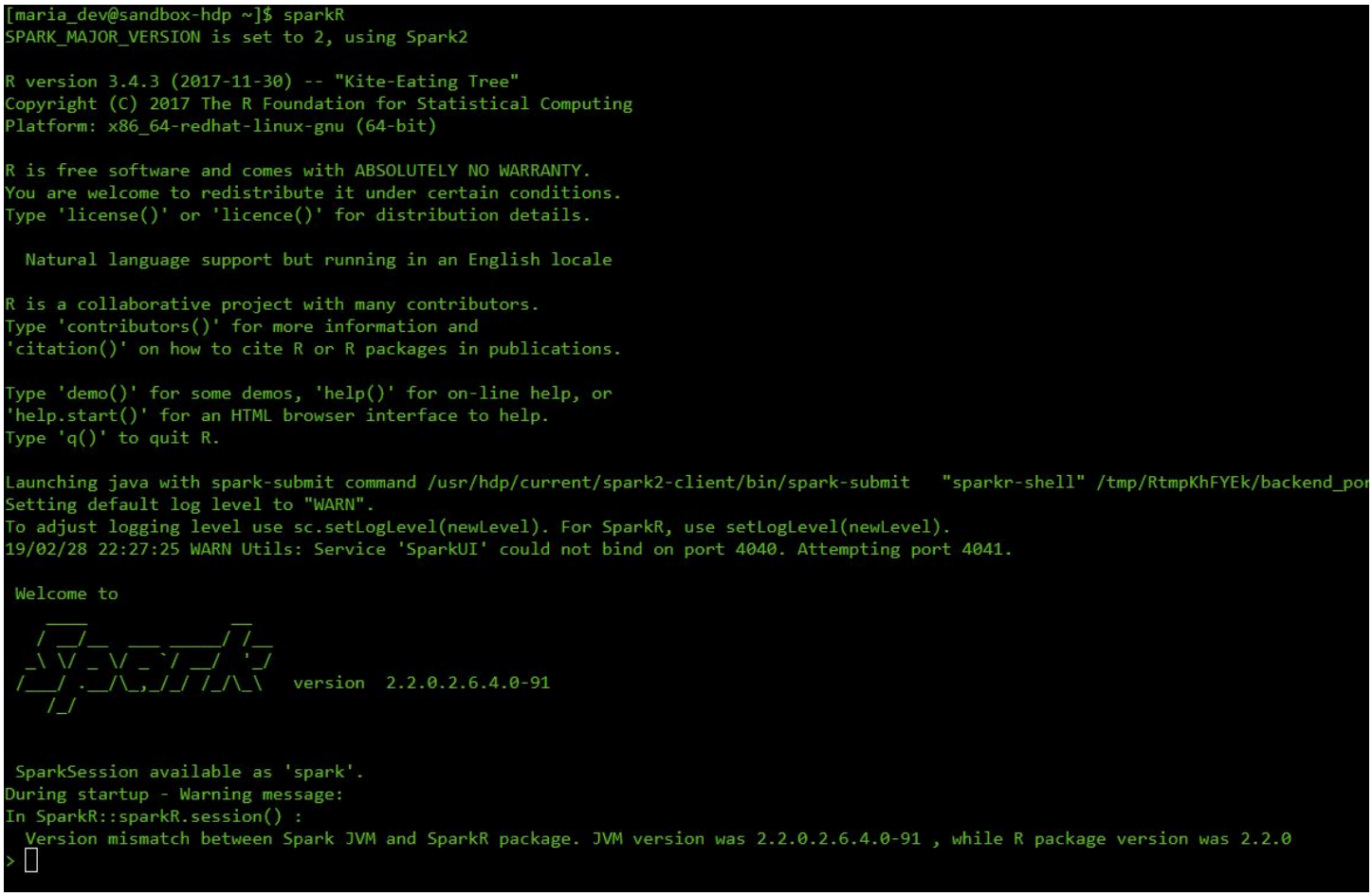
Your output is likely to differ because your reducers are unlikely to do the exact same thing my reducers did.

g. When you are finished, simply close out of the Zeppelin web page. Your notebook will be automatically saved. When you return, simply click on the Notebook name on the left hand side to open it back up.

6. Using Scala on Spark vs using PySpark is generally just a personal preference. Scala is faster than Python on Spark as Spark is written in Scala. As you have seen in the previous steps, the difference between Scala code and Python code is relatively small. SparkR supports operations like selection, filtering and aggregation on large datasets. The following steps will walk you through some of the most common things that you'll need to know in order to run a SparkR program.

   a. Unfortunately, as of this writing, the Zeppelin Notebook does not connect well with the SparkR interpreter. There are ways to connect RStudio to your Hortonworks machine but the process is non-trivial and not worth the effort. Our goal is to simply learn some of the commands that SparkR offers. We are less interested in the administrative side of setting up the environment. Therefore, all of our commands will be written using the terminal window. Connect to http://yourAzureIP:4200 and enter in maria_dev as the username and password.

b. Enter in **sparkR** to start up the interpreter. You should see something like this:



The following steps will guide you through some of the common commands you can use with SparkR. Many of the commands will be self-explanatory. An explanation will only be given if it is not obvious what is going on. We will be creating a DataFrame with our taxi data.

c. **`taxi <- read.df("/user/zeppelin/taxi.csv", "csv", header = "true", inferSchema = "true")`**

d. **`head(taxi)`**

Prints out the first few rows of the dataframe.

e. **`only_passengers <- select(taxi, "passenger_count")`**

f. **`head(only_passengers)`**

g. `small_fares <- filter(taxi, taxi$fare_amount < 10)`

   Filters the taxi DataFrame to only include fares that are less than $10. If we wanted to show the first few rows, we could say **`head(small_fares)`**. If we wanted to know how many there were, we could say **`count(small_fare)`**.

h. `head(summarize(groupBy(taxi, taxi$passenger_count), how_many = n(taxi$passenger_count)))`

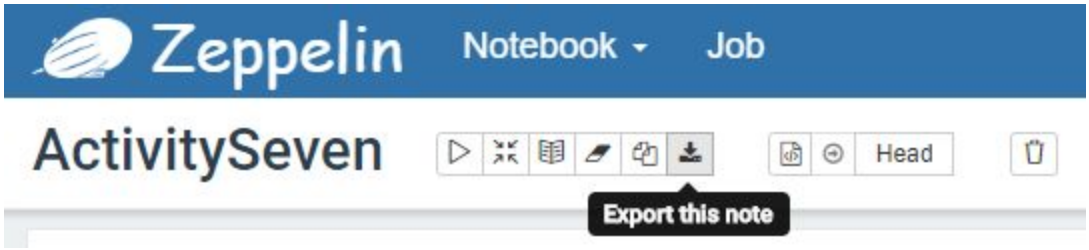
   The groupBy command does what one would expect: it groups the rows by passenger_count. The summarize function takes two arguments in this example. The first is the DataFrame that we have grouped together. The second is how we wish to summarize it. In other words, we wish to sum up how many passengers were in a particular grouping. This is what the n(...) function does. In the taxi example, 1 passenger was the most common by far with 7,898,295 rides. If we wish to see the counts in descending order, we use the following commands:

i. ```
   totals <- summarize(groupBy(taxi, taxi$passenger_count),
   how_many = n(taxi$passenger_count))
   head(arrange(totals, desc(totals$how_many)))
   ```

j. `taxi$fare_plus_tip <- taxi$fare_amount + taxi$tip_amount`

   This command allows us to easily add columns to our DataFrame. Doing so updates the original DataFrame.

k. `taxi_updated <- drop(taxi, "tolls_amount")`

   If we wish to drop a column, we need to assign that to a new DataFrame.

l. SparkR can also be used to run SQL queries. The following set of commands are how you setup your DataFrame such that you can run any SQL query you want on your data:

   ```
   createOrReplaceTempView(taxi, "taxi_for_sql")
   large_fares <- sql("SELECT * FROM taxi_for_sql WHERE
   fare_amount > 100")
   head(large_fares)
   ```
   ```

SparkR also supports many machine learning algorithms including classification, regression, clustering, etc. Such algorithms are beyond the scope of this course but if you are interested, you can read all about them here: https://spark.apache.org/docs/2.2.0/sparkr.html#machine-learning

If you are particularly interested in a specific algorithm, feel free to ask about it on the message board and we'll do the best we can to answer any questions you have about it.

7. **For your submitted work**, solve the following problems. Create a new note in Zeppelin and call your note ActivitySeven. Use spark2 as the default interpreter. You should create 3 paragraphs, 1 for each problem. When you are finished, click on the **export this note** option:



Upload the ActivitySeven.json file to the dropbox.

For each problem, all words are separated by whitespace (spaces, tabs, newlines) only. All other characters belong to a word. For example, **don't** is a word and **hello,** is a word (note the comma) which is different from the word **hello** that might appear in the document. A word must contain at least 1 character (i.e. the empty string is ignored for all problems). For each problem, the words should be case-insensitive: *the* and *The* are the same word.

a. Print out all words that appear between 5 and 7 times (including 5 and 7) in the wap.txt document. Each word should be separated by a newline. The wap.txt file must be located in this directory: /user/zeppelin/ and you can access the wap file by using this command: **val wapFile = sc.textFile("wap.txt");**

b. Print out the vowel combination that appears in the most words in the wap.txt document (place in same directory as above). A combination of vowels can contain any number of vowels (even 0 vowels or 1 vowel). A vowel is one of {a, e, i, o, u}. For instance, if the document were:
This is a tester file ice exit.
There are 2 words that contain a single *i*, there is 1 word that contains a single *a*, there is 1 word that contains two *e's* and there are 3 words that contains an *e* and an *i*. In this case, *ei* would print out as the answer because that combination of vowels appears in three different words. The answer to this should not surprise you. In my tests, I found a vowel combination that appeared in over 85,000 words.

c. Consider the taxi dataset from step 3. Which grouping of fares tips the best? The fare groups you should consider are this: $0 - $24.99, $25 - $49.99, $50 - $74.99, $75+. When considering the best tip, you should consider the tip as a percentage of the fare.

d. Consider the taxi dataset from step 3. What were the top 10 fare_amount / trip_distance amounts? In other words, what trips cost the most for the least amount of distance traveled? For

this problem, you should not consider any trip_distances with a value of 0 as all of them would have an infinite fare to distance ratio.

The answer to the question in task 1:

```
val answer = wordsFile.reduce((first, second) => if(first.split(" ").size > second.split(" ").size) first else second).split(" ").size;
```

A solution to the question in task 4:

```
answer.createOrReplaceTempView("answerTable")
spark.sqlContext.sql("SELECT * FROM answerTable ORDER BY someAverage DESC")
```

It appears that 4:25am is the best hour to start working. 4:26am and 4:37am are the next best options. A conjecture for this might be that people who leave that early in the morning are doing so for two reasons: to go to the airport or to go a long way to work. The airport is likely further away than normal and if you are leaving that early, then work is probably further away than average. In an attempt to justify this claim, I looked at the lowest fares by changing DESC to ASC. The lowest fares start at 6:57am and continue with 6:58am and 6:53am. People who leave at that time are likely closer to work, hence the lower fare. People who are going to the airport at that time aren't necessarily leaving on a much later flight, rather, they probably live closer to the airport and can leave later. One could likely determine if this claim is true by using the pickup and dropoff locations in the dataset. It would be an interesting question for the final project.

You can ignore everything after this line. It is work in progress.

EXTRA Scala stuff, not terribly useful for now. Also some SparkR at the end. Trying to make it work with Zeppelin on Hortonworks.

a. **val numbers = Array(1, 2, 3, 4, 5);**
   i. A simple array to demonstrate how to fold.

b. **numbers.fold(0){(a,b) => a+b}**
   i. A fold is a more generic version of a reduce. The difference between this and step 1e is that you give a fold an initial value, the (0). Initially, a == 0 and b == 1 because 1 is the first element of the Array numbers. After the first addition, a becomes 1 (0+1) and b becomes 2 because it is the second element of numbers. Then a becomes 3 (1+2) and b becomes 3 because it is the third element and so on.

c. **import org.apache.spark.rdd.RDD**
   def printOff(data: RDD[_], number: Int = 5): Unit = {
   println("RDD is a: " + data)
   data.take(number).foreach(println)
   }
   i. There must be a newline between the *println("RDD is a: " + data)* statement and the *data.take(number).foreach(println)* statement. If you do not put a newline between them, you must add a semicolon between the statements.
   Scala is not a pure functional language. A pure functional language does not allow for mutable variables. However, Scala is able to do many of the procedural statements that you are used to in Python. The above is a simple function that prints out the first *number* elements of an RDD called *data*. Since we are using an RDD in this function, we need to import it:
   import org.apache.spark.rdd.RDD
   The function definition is similar to what you see in Python. However, with Scala, we need to define the type that the function is going to return. In this case, we aren't returning anything, so we simply use Unit. The *printOff* is just a name I chose for the function. *data* and *number* are the parameters to this function and *RDD[_]* and *Int* are their respective types. The thing after RDD is a left brace [followed by an underscore _ and completed by a right brace]. The underscore is essentially a wildcard. This function will accept an RDD of any type. The *= 5* is the default value for number. If a value is not provided, it's default value is 5.
   The print statement is fairly self-explanatory. data.take(number) is saying that we want to retrieve the first 5 elements of the RDD. The foreach call is saying that

we want to applying some function to those elements. The println is the function that we are going to apply to those elements. A Python equivalent for this function would look something like this:

```
def printOff(data, number = 5):
  for i in range(1,number+1):
    print(data[i])
```

d. **printOff(wordsFile);**
   i. This is a call to the function we just created. Since number had a default value of 5, we didn't need to specify how many elements we wanted to print out. If we wanted to specify the first 20 lines, we would do something like this: **printOff(wordsFile, 20)**

e. **def isVowel(letter: Char): Boolean = {**
```
 letter match {
  case 'a' => true;
  case 'e' => true;
  case 'i' => true;
  case 'o' => true;
  case 'u' => true;
  case _ => false;
 }
}
```
   i. A simple example showing off how a match/case statement works. A match followed by a bunch of cases is essentially a lot of if statements. If the letter matches any vowel, the function returns true, otherwise it returns false. This function also shows off how to return a value. In this function, we are returning a Boolean. This function will likely be helpful for problem 5b.

f. **val groupThem = flattenMap.groupBy(word => word.size);**
   i. The *groupBy* function is a nice SQL-like function that allows us to group things. In this case, we are simply grouping them by the actual word size.

g. **val mapThem = groupThem.map(groupThem => (groupThem._1, groupThem._2.size));**
   i. An a reminder, groupThem is an RDD[(Int, Iterable[String])]. When referring to _1 in this statement, we are referring to the first element of the RDD, i.e. the Int. _2 represents the Iterable[String].

h. **val someMap = List(("first", 20), ("second", 15), ("third", 10));**
val someMapRDD = sc.parallelize(someMap);
    i. The sc.parallelize is done in order to test an RDD without having to read in from a file. The type of someMapRDD will be an RDD[(String, Int)].

i. **val someMapTwo = List(("first", 26), ("second", 12), ("fourth", 6));**
val someMapTwoRDD = sc.parallelize(someMapTwo);
val someJoinedMap = someMapRDD.join(someMapTwoRDD);
someJoinedMap.collect();
    i. The two RDD's will be joined together to create an RDD[(String, (Int, Int))]. A join is done using the first field. In this example, the "third" string cannot be joined to anything in the second RDD and therefore it does not appear in the final someJoinedMap. Similarly, "fourth" does not appear in the right RDD so it does not appear in the final someJoinedMap. One can use leftOuterJoin or rightOuterJoin respectively to include those elements.

j. **var changeable = Array(1,2,3);**
    i. Everything up until this point has been a val. A val is an immutable variable. However, a var is a variable that can get a new value. You should use var's sparingly though. Scala is meant to be a functional language and creating mutable variables is not the best idea. If we want to change our variable to be something else, this is allowed:

    changeable = Array(4,5,6)
    changeable(1) = 10
    changeable

    The above prints out Array(4,10,6)

k. **val numbers = Array(1,2,3,4,5)**
for {value <- numbers} yield { value + 5 }
for {word <- flattenMap} yield {word.toUpperCase}
    i. A less than ideal way to iterate through an array or some RDD. The generic code looks like:
    for{element <- listOrRDD} yield { something }
    Where the something is whatever you want to change element to. A *for loop* is essentially a map and you should use a map instead of a loop. This is being shown for informational purposes only. The above code could have also been accomplished by using:

numbers.map(value=>value+5)
flattenMap.map(word=>word.toUpperCase)

l. **var total = 0**
val numbers = Array(1,2,3,4,5)
for {value <- numbers} {total = total + value}
   i. A more procedural like sequence of statements. We have a variable called total which we update whenever we loop through an array of numbers.

2. OLD QUESTION ON SPARK:

   a. Create a function called **similarWord** that accepts two filenames as String arguments and prints out one word.
   b. The uncommon word that is shared the most. Let's say fileOne(*word*) is the number of times *word* appears in fileOne and fileTwo(*word*) is the number of times *word* appears in fileTwo. This function should print the uncommon word that maximizes this function:
   min(fileOne(word), fileTwo(word))

   What this will do is allow us to find the word that is important to both files. If *California* appears 1000 times in one file and 1 time in another file, the value of that function is 1: min(1000,1). Which makes sense… California only appears once in the other file so it isn't shared that much. However, if *Wisconsin* appears 38 times in one file and 45 times in another file, the value of that function is 38: min(38,45). Since it appears in both a high number of times, it is shared well between the two files. This is one way we could build a list of topics that two documents share to see how similar one file is to another.
   It is likely that *the* or *and* is the most shared word between any pair of documents. Therefore, we want to filter out common words. Download http://www.uwosh.edu/faculty_staff/krohne/ds730/common.txt and use it to ensure that the word shared the most between the two files is actually an uncommon word (i.e. not in the common.txt file). Be sure the common.txt file is stored in the /user/zeppelin/ folder for easy access. A few clarifications are below:
      i. Do not worry about error checking the filenames that are passed in. The filenames I test will exist.
      ii. Do not strip punctuation from the words. In other words, **the:** and **the** are two different words.
      iii. If there is a tie between uncommon words that appear the most often, print them both out.