```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q https://apache.mirror.colo-serv.net/spark/spark-2.4.7/spark-2.4.7-bin-hadoop2.7.t
!tar xf spark-2.4.7-bin-hadoop2.7.tgz
!pip install -q findspark

import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.7-bin-hadoop2.7"

import findspark
findspark.init("spark-2.4.7-bin-hadoop2.7")# SPARK_HOME

import pyspark
from pyspark.sql import *
from pyspark.sql.functions import *
from pyspark import SparkContext, SparkConf

sc = SparkContext.getOrCreate()
spark = SparkSession.builder.getOrCreate()
```

Create an RDD from a text file

Each line of the text file becomes an element of the RDD.

```
!wget http://www.gutenberg.org/files/2600/2600-0.txt -O war_and_peace.txt
textFile = sc.textFile('war_and_peace.txt')
```

```
--2021-03-17 01:59:58--  http://www.gutenberg.org/files/2600/2600-0.txt
Resolving www.gutenberg.org (www.gutenberg.org)... 152.19.134.47, 2610:28:3090:3000:0:ba
Connecting to www.gutenberg.org (www.gutenberg.org)|152.19.134.47|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3359584 (3.2M) [text/plain]
Saving to: ‘war_and_peace.txt’

war_and_peace.txt   100%[===================>]   3.20M  5.01MB/s    in 0.6s

2021-03-17 01:59:59 (5.01 MB/s) - ‘war_and_peace.txt’ saved [3359584/3359584]
```

```
#One common transformation is
#filtering data that matches a predicate.
#We can use this to create a new RDD
#holding just the strings that contain
#the word Anna.

# The filter() transformation returns a new RDD
# containing only the elements that satisfy a predicate.
# A predicate is a function that returns True or False
```

```
# given an element of the RDD.
# The following function "lambda x: "Anna" in x",
# given an element x of the RDD, a line in this case,
# returns condition '"Anna" in x', which can be True or False.
annaLines = textFile.filter(lambda x: "Anna" in x)

#One example of an action is first()
#which returns the first element in an RDD.
firstLine = annaLines.first()

print(firstLine)

#Another example of action is collecting
#all the elements of an RDD.
allAnnaLines = annaLines.collect()

print(allAnnaLines)
```

```
It was in July, 1805, and the speaker was the well-known Anna Pávlovna
['It was in July, 1805, and the speaker was the well-known Anna Pávlovna', 'rank and imp
```

```
#RDD<String>
N = textFile.filter(lambda x: "Natasha" in x)
#RDD<String>
P = textFile.filter(lambda x: "Pierre" in x)
#RDD<String>
A = textFile.filter(lambda x: "Andrew" in x)
#RDD<String>
NaP = N.intersection(P)
#RDD<String>
NaA = N.intersection(A)
#RDD<String>
NaPoNaA = NaP.union(NaA)

#These RDD have only been defined; they are not computed yet.
#They will only be computed if some action is performed as follows.

print(len(NaP.collect()))
print(len(NaA.collect()))

print(NaPoNaA.collect())
```

```
0
0
[]
```

```
#map() takes in a function and applies it to each element in the RDD
#with the result of the function being the new value of each element
#in the resulting RDD.
```

```
rdd = sc.parallelize([1, 2, 3, 4]);
result = rdd.map(lambda x: x*x);
print(result.collect());
```

```
[1, 4, 9, 16]
```

```
#Sometimes we want to produce multiple output elements for each input element.
#The operation to do this is called flatMap().
#As with map(), the function we provide to flatMap() is called individually
#for each element in the input RDD.
#Instead of returning a single element, we return in this function an iterator
#with our return values.
#Rather than producing an RDD of iterators, flatMap() gives back an RDD
#of the elements from all of the iterators.

#A simple usage of flatMap() is splitting up an input string into words.
#From each line, we want to output multiple words.

words = textFile.flatMap(lambda x: x.split());

print(words.collect()[0:100])
print(words.count())
```

```
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'War', 'and', 'Peace,', 'by', 'Leo', 'To
566316
```

```
#Suppose we would like to transform our string RDD of words
#to an RDD of the word lengths so that we can compute different stats with ease.

wordLength = words.map(lambda x: len(x));

#Then, we can compute different stats on it. E.g.

wordAvgLength = wordLength.mean();

print(wordAvgLength)

#and quite a few others (min, max, stdev, histograms, etc).
print(wordLength.max())
```

```
4.669703840258791
36
```

```
#The most common action on basic RDDs you will likely use is reduce(),
#which takes a function that operates on two elements of the type in your RDD
#and returns a new element of the same type.

#A simple example of such a function is +, which we can use to sum our RDD.
#With reduce(), we can easily sum the elements of our RDD.
```

```
#With reduce(), we can easily sum the elements of our RDD,
#count the number of elements, and perform other types of aggregations.

rdd = sc.parallelize([1, 2, 3, 4]);
sum = rdd.reduce(lambda x,y: x+y);
print(sum)
```

```
10
```

```
#reduce() requires that the return type of our result be the same type as that
#of the elements in the RDD we are operating over.
#This works well for operations like sum,
#but sometimes we want to return a different type.

#For example, when computing a running average,
#we need to keep track of both the count so far and the number of elements,
#which requires us to return a pair.
#We could work around this by first using map() where we transform every element
#into the element and the number 1, which is the type we want to return,
#so that the reduce() function can work on pairs.

rdd = sc.parallelize([1, 2, 3, 4])
sumcnt = rdd.map(lambda x: (x,1) ).reduce(lambda t,r: (t[0]+r[0], t[1]+r[1]) )

avg = sumcnt[0] / sumcnt[1]
print(avg)
```

```
2.5
```

```
#The aggregate() action frees us from the constraint of having the return
#be the same type as the RDD we are working on.
#With aggregate(), we supply:
#(1) An initial “zero” value of the type we want to return.
#(2) A function to combine the elements from our RDD with the “accumulator”.
#(3) A second function to “merge” two accumulators,
#    given that each machine accumulates its own results locally.

#We can use aggregate() to compute the average of an RDD,
#avoiding a map() before the reduce().

rdd = sc.parallelize([1, 2, 3, 4])
sumcnt = rdd.aggregate((0, 0),
                       lambda acc, value: (acc[0] + value, acc[1] + 1),
                       lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])  )
avg = sumcnt[0] / sumcnt[1]
print(avg)
```

```
2.5
```

RDDs of key/value pairs

Spark provides operations on RDDs containing key/value pairs. These RDDs are called pair RDDs. Pair RDDs allow you to act on each key in parallel. For example, pair RDDs have a reduceByKey() method (analogous to reduce for regular RDDs) that can aggregate data separately for each key. We can create pair RDDs from existing RDDs. E.g.

```
import re
words = textFile.flatMap(lambda x: re.findall('\w+', x));

lw = words.map( lambda x: (len(x), x) );

# This creates an RDD of length-word pairs.
# What can we do with it?
# We can find for example the number of words for each length.

r = lw.countByKey();
print(r)

# Or, we can collect all the words of length >= 16.

longwordsRDD = lw.groupByKey().filter(lambda x: x[0] >= 16)

print(longwordsRDD.collect())

#What we get back is an object which allows iterating over the results.
#Turn the results of groupByKey into a list by calling list() on the values, e.g.

print(longwordsRDD.map(lambda x : (x[0], list(x[1]))).collect())
```

```
defaultdict(<class 'int'>, {3: 143657, 7: 42864, 9: 17486, 5: 58450, 2: 95460, 4: 98716,
[(16, <pyspark.resultiterable.ResultIterable object at 0x7f1ab6a402d0>), (18, <pyspark.r
[(16, ['enthusiastically', 'circumstantially', 'incomprehensible', 'misunderstanding',
```

Word count

```
textFile = sc.textFile('war_and_peace.txt')

word_counts = textFile.flatMap(lambda x: x.split()) \
                      .map(lambda word: (word,1)) \
                      .reduceByKey(lambda a,b: a+b)

print(word_counts.collect())

# Those familiar with the combiner concept from MapReduce should note that
# calling reduceByKey() will automatically perform combining locally
# on each machine before computing global totals for each key.
# The user does not need to specify a combiner.
```

```
[('The', 2550), ('Project', 78), ('EBook', 2), ('of', 14857), ('Peace,', 2), ('Leo', 4),
```

Word count with stopwords removed

```
!wget "https://gist.githubusercontent.com/sebleier/554280/raw/7e0e4a1ce04c2bb7bd41089c9821

textFile = sc.textFile('war_and_peace.txt')
stopwords = sc.textFile('stopwords.txt')

word_counts = textFile.flatMap(lambda x: x.split()) \
                      .map(lambda word: (word.lower(),1)) \
                      .subtractByKey(stopwords.map(lambda word: (word, 1))) \
                      .reduceByKey(lambda a,b: a+b)

print(word_counts.collect())
```

```
--2021-03-17 02:00:14--  https://gist.githubusercontent.com/sebleier/554280/raw/7e0e4a1c
Resolving gist.githubusercontent.com (gist.githubusercontent.com)... 185.199.108.133, 18
Connecting to gist.githubusercontent.com (gist.githubusercontent.com)|185.199.108.133|:4
HTTP request sent, awaiting response... 200 OK
Length: 622 [text/plain]
Saving to: ‘stopwords.txt’

stopwords.txt       100%[===================>]     622  --.-KB/s    in 0s

2021-03-17 02:00:14 (30.3 MB/s) - ‘stopwords.txt’ saved [622/622]

[('gutenberg', 24), ('leo', 4), ('tolstoy', 3), ('whatsoever.', 2), ('may', 260), ('it,
```