Chapter 2
Working with PySpark

1. Spark Capabilities

Spark is capable of doing almost everything you need to process and work with big data sets. It achieves scalability through its built-in nature and can work with scripts written in Scala, Java and Python. On top of it, you may wish to use different libraries to conduct your work.

SparkStreaming: This is used to work with the data set that updates frequently or near real-time or real-time. For example, weblog or social media or smart meter data that is updated (or) changed over time very fast.

SparkSQL: This enables running Spark on top of Hive context and is generally used to make queries for data subjects/objects/variables on structured data. Essentially, this is SQL (sequel) in Spark context for structured data (data warehouse, databases).

MLLib: This library contains a set of machine learning algorithms. You can use this library for statistical analysis of your RDD. Most of the statistical output can be derived by using this library.

GraphX: This is particularly useful for any graphical analysis based on graph theories and is generally used to extract high level information on networks, say, for example, social network to know how people are linked and how they communicate with each other.

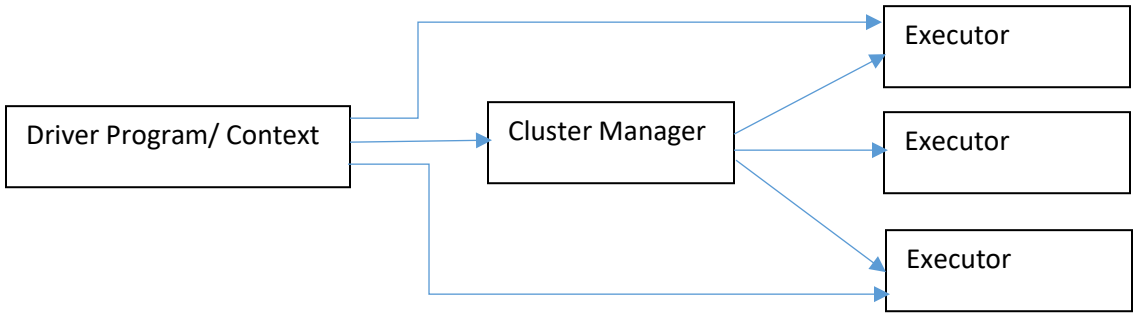
2. Spark with Python

There are three different choices for scripting languages: Python, Scala, and Java. Using Python has some good favors for analysts and data scientists:

(1) Less coding required: only few lines of codes can do lots of work for you.
(2) No dependency requirement for compiling and running scripts
(3) Built-in libraries are ready for use
(4) Though Scala is native to Spark, you can do similar task in Python. Further, there are lots of similarities between Scala and Python when used with Spark.

3. How Does This Work?

It determines workflows and optimize workflows by using DAG (directed acyclic graph) controller/ engine. It's work type cane explained by the following graph. It creates a Spark Context as part of Driver Program (sometimes it is called Master). This Driver Program (or Master) communicates with Executors (also known as Executor Nodes/ Slaves/ Cores) via a Cluster Manager.



As can be seen in the above figure, both driver program and cluster manager can form and distribute work in different nodes (or cores of your computer). The beauty of Spark is that it is not only scalable but also fault tolerant. By fault tolerance we mean that when a node (or an executor or a core) stops working (or goes down), the entire execution process does not go down: It keeps working and at some point either recovered or passed to the next available node (slave or core) that becomes take on the job. In fact, it scales up to entire cluster of computers (or CPU cores in your computer when use as standalone mode). But as a programmer you will see it working by running scripts in your own computer in usual way, and we explore this by running a simple script in next sections.

3.1 Creating Resilient Distributed Data (RDD)

Driver program works through a context to proceed with RDD objects and this is the core object the Spark system works with. Essentially, this refers to data set, an abstraction of a giant set of data. Though RDD is resilient and distributed, and can work/spread across entire cluster of computers (or nodes or cores of your computer). We need to set up of RDD objects and load them with big data sets. Next we call various methods/functions to work with these RDD objects for distributed processing of data.

This may or may not work locally, but can handle with failure of any working nodes/cores automatically. If one of the core/node shuts down, this can be recovered or can be passed to the other nodes to get the job done. This is one of the most beautiful advantage of Spark over other computing environment, say, for example, with Hadoop ecosystem.

Speed is also extremely good. It is 100x faster than Hadoop MapReduce for in-memory processing and around 10x faster when processing is done in disks.

With a lot of flexibility of computational tools, you may use either Python or Scala or Java as your scripting language. It is always good to learn more than one scripting language. But if you learn Python carefully, I believe you can do almost everything with Spark. Though Scala is the native language of Apache Spark (Spark is built on Scala), Python in Apache Spark works in a similar fashion. So, if you are good at Python, then you can utilize full power of Apache Spark. I would prefer Python, because it will support for all sort of machine learning, predictive and graphical models with existing Python library functions.

3.2 Creating Spark Context

Let us come back to the concept RDD. As a developer and data scientist/analyst, you only need to know the data environment to change the data structure from one environment to the other (one structure to the other). So, we create a context for data to transfer with respect to its structure. This is the Spark Context ("sc" is used in coding to refer this) object that we create to do all operation in Spark.

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("EnergyRating")
sc = SparkContext(conf = conf)
```

At the beginning of the constructing a Spark Context, we should import Spark libraries for Python (as can be seen in the above snippet): from pyspark import SparkConf, SparkContext where SparkConf is used to do configuration and SparkContext is used to assign context based on that configuration.

As can be seen in the snippet, configuration is set to "local" not to "cluster", this means that the context is designed to work with a single core of local machine (computer). The setAppName assigns the task name by which we understand the type of work is going to be done in the context (for example, working in the context of EnergyRating as shown in the snippet). Once this configuration and context settings are done, you are now ready to create RDD. For example,

RDD = sc.textFile("README.md")

creates a context that reads text data from README.md file. To create a RDD from Spark context use the command

datalines = sc.textFile("file:///c:/SparkPythonCourse/energyrating.txt")

This abovementioned context is suitable only when you have a data file that is stored on disk and can be accessed by assigning a path to the file and the file is not too big to fit in the disk (simply not big enough, not a big data). However, this can be done for other data types, for example, to access distributed data we use

s3n:// and hdfs:// (are used for distributed data sets when they do not fit within the machine)

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("WarrantyClaim")
sc = SparkContext(conf = conf)

datalines = sc.textFile("file:///SparkPythonCourse/warrantyclaim.txt")
```

In this snippet shown above, we see that a Spark Context (sc) has been defined and later an RDD (here, datalines) is created by using the "sc". The RDD is an object that reads (contains) lines (each row is considered as a line). This energyrating.txt file contains rows with information related to item id, product code, energy rating (star), warranty period, product life, and claim status (free text).

1	358625461	12	1.5	9	Refund
2	458625452	24	3.5	18	Replace
3	459625460	12	2.5	10	Refund
4	358625472	24	4.5	22	Repair
5	458725461	24	4.0	17	Replace

So the RDD (datalines) contains 10K lines shown above (not big enough, but good for a handy example), where the first value of this RDD is the first line of text, second value is the second line of the text, and so on. Each text line is then considered as a single string with some white spaces inside. In order to work with this, we need to do some transformation of this RDD.

3.3 Transformation of RDD

RDDs can be transformed from one RDD to the other and this is made possible by using some buit-in functions. Some example functions are:

- **map**: This gets some data and transforms to another set of data. This function transforms data ensuring a one-to-one relationship with previous RDD.
- **filter:** This function filters out information (keyword, message, error message, etc.) from the data. Thus after throwing out uninteresting information, this produces another RDD.
- **distinct:** This is used to obtain an RDD with distinct/unique values (properties).
- **union:** To obtain union of two RDDs
- **intersection:** Two obtain intersection of two RDDs
- **subtract:** Subtracts one RDD from the other.
- **sample**: To obtain a random sample from the RDD to construct another set of RDD (data) with size smaller than the original one.

3.4 Actions on RDD

Nothing happens, until an action is performed. Spark becomes ready by constructing a DAG for workflow and wait for an action to complete the task. When an action is called, it gets to work. This behavior is known as Lazy Operation in Spark. Some actions command could be like these:

count: This produces the number of elements in the RDD and the following action produces a result 3.

```
sc.parallelize([1, 2, 3]).count()
```

countByValue: Count by unique values, the following action produces results like (number, number of times) as here in

(1, 3)

(2, 2)

```
sc.parallelize([1, 2, 1,1,2]).countByValue()
```

collect: This returns a list of elements in the RDD, for example, the following action produces [1, 2, 3]

```
sc.parallelize([1, 2, 3]).collect()
```

reduce: may use combinations of operations and mapping to produce results for program driver

```
sc.parallelize([1, 2, 3]).reduce(lambda a,b: a+b)
```

take: takes specified number of values from the beginning of an RDD, for example, the following action takes the first element in the RDD and produces the result 1 for the first action, [1, 2] for the second action, and [1, 2, 3] for the third action.

```
sc.parallelize([1, 2, 3]).take(1)

sc.parallelize([1, 2, 3]).take(2)

sc.parallelize([1, 2, 3]).take(10)
```

ActionOnRDD.py

```
1  from pyspark import SparkConf, SparkContext
2
3  conf = SparkConf().setMaster("local").setAppName("LoanDataPractice")
4  sc = SparkContext(conf = conf)
5
6  def fprint(x): print(x)
7
8  rdd1 = sc.parallelize([1, 2, 3])
9
10 print("Print each element of RDD1:")
11 rdd1.foreach(fprint)
12
13 print("Counting Value in RDD1: ", rdd1.count() )
14
15 rdd2 = sc.parallelize([1,2,1,1,2])
16 print("Counting Value in RDD2: ", rdd2.count() )
17 print("Count by Value in RDD2: ", rdd2.countByValue() )
18
19 print("Reduce to Sum of Values in RDD1: ", rdd1.reduce( lambda x,y: x + y ) )
20 print("Take the First Value from the Beginning in RDD1: ", rdd1.take(1) )
21 print("Take the First 2 Values from the Beginning in RDD1: ", rdd1.take(2) )
22 print("Take the First 10 Values from the Beginning in RDD1: ", rdd1.take(10) )
```

```
(User) c:\SparkPythonCourse\programs>spark-submit ActionOnRDD.py
Print each element of RDD1:
1
2
3
Counting Value in RDD1:  3
Counting Value in RDD2:  5
Count by Value in RDD2:  defaultdict(<class 'int'>, {1: 3, 2: 2})
Reduce to Sum of Values in RDD1:  6
Take the First Value from the Beginning in RDD1:  [1]
Take the First 2 Values from the Beginning in RDD1:  [1, 2]
Take the First 10 Values from the Beginning in RDD1:  [1, 2, 3]
```

3.5 Transformation + Action

Transformation refers to the operation applied on a RDD to create new RDD. Whereas, actions refer to an operation which also applies on RDD, that instructs Spark to perform computation and send the result back to driver.

Mapping as transformation: An example of this is assign data to construct RDD and then use map function for transformation.

```
RDD1 = sc.parallelize([1,2,3])
RDD2 = RDD1.map( lambda x: x+2)
```

So, RDD1 contains values 1,2,3 but RDD2 contains values 3,4,5.

Sometimes the function, like the lambda function above, may not be simple and in that case we may define a separate function and call that function to do a similar sort of mapping operation. For example,

```
def add2(x):
        return x+2
RDD2 = RDD1.map(add2)
```

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("WarrantyClaim")
sc = SparkContext(conf = conf)

datalines = sc.textFile("file:///SparkPythonCourse/warrantyclaim.txt")

WarrantyPeriod = datalines.map(lambda x: x.split()[2])
ProductLife = datalines.map(lambda x: x.split()[4])
EnergyRating = datalines.map(lambda x: x.split()[3])

def ShortLife (x):
    return x.split()[2] - x.split()[4]
LifeShortage = datalines.map(ShortLife)
```

In the above snippet, we used the Warranty Claim data. This mapping process transforms the RDD by splitting the text by whitespace to individual field and then by assigning the third field/split to WarrantyPeriod (new RDD). Similar transformation happens with ProductLife and EnergyRating. We can also define the function as has been defined earlier and can use this too as in the snippet. Thus the LifeShortage is a RDD is

3
6
2
2
7

Counting by value for action: For example,

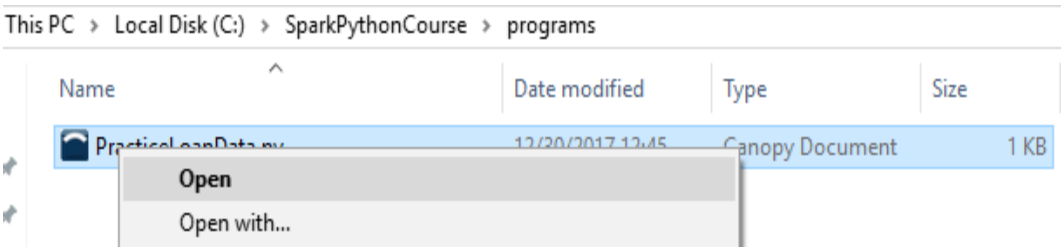
```
LifeCount = LifeShortage.countByValue()
```

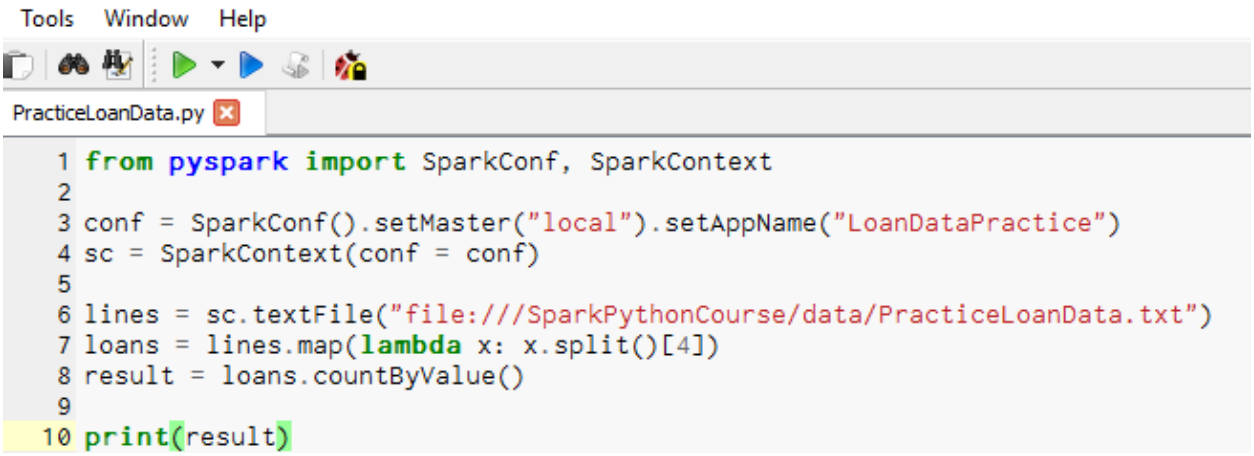
provides value pairs

(3, 1)
(6, 1)
(2, 2)
(7, 1)

3.6 Running a Python Script

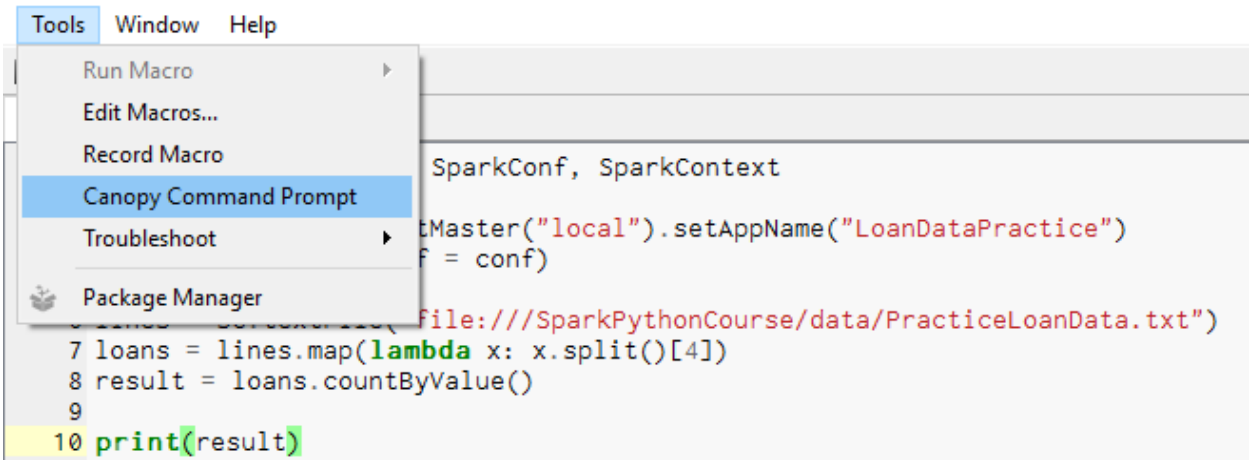
Let us locate an example file considered for this course: the file is in C:/SparkPythonCourse/Programs and the file name is PracticeLoanData.py.





Double click on the file and this will open a window like shown below. Now trace the location of your data file in the computer and change the path to the data file. The text file contains a data snippet (age, sex, household income, number of dependents, car loan, property ownership, group) to check manual work with coding works.

44	M	57000	0	No	Rent	2
35	F	56000	1	Yes	Own	1
39	F	76000	2	Yes	Rent	2
53	M	65000	1	Yes	Rent	1
43	F	45000	3	No	Own	2
37	M	49000	1	No	Own	2



Administrator: Canopy Command Prompt

```
(User) c:\Users\tumpa>cd c:\SparkPythonCourse\programs

(User) c:\SparkPythonCourse\programs>dir
 Volume in drive C has no label.
 Volume Serial Number is 4C3B-196B

 Directory of c:\SparkPythonCourse\programs

12/30/2017  12:55 AM    <DIR>          .
12/30/2017  12:55 AM    <DIR>          ..
12/30/2017  12:45 AM               318 PracticeLoanData.py
               1 File(s)            318 bytes
               2 Dir(s)  30,634,967,040 bytes free

(User) c:\SparkPythonCourse\programs>spark-submit PracticeLoanData.py
defaultdict(<class 'int'>, {'No': 3, 'Yes': 3})

(User) c:\SparkPythonCourse\programs>
```

PracticeLoanData.py

```
 1 from pyspark import SparkConf, SparkContext
 2
 3 conf = SparkConf().setMaster("local").setAppName("LoanDataPractice")
 4 sc = SparkContext(conf = conf)
 5
 6 lines = sc.textFile("file:///SparkPythonCourse/data/PracticeLoanData.txt")
 7 loans = lines.map(lambda x: x.split()[4])
 8 result = loans.countByValue()
 9
10 for key, value in result.items():
11     print("%s %i" % (key, value))
```

Administrator: Canopy Command Prompt

```
(User) c:\SparkPythonCourse\programs>spark-submit PracticeLoanData.py
No 3
Yes 3

(User) c:\SparkPythonCourse\programs>
```

3.7 Some Common Transformations

Filtering is done by using a lambda function in pyspark.

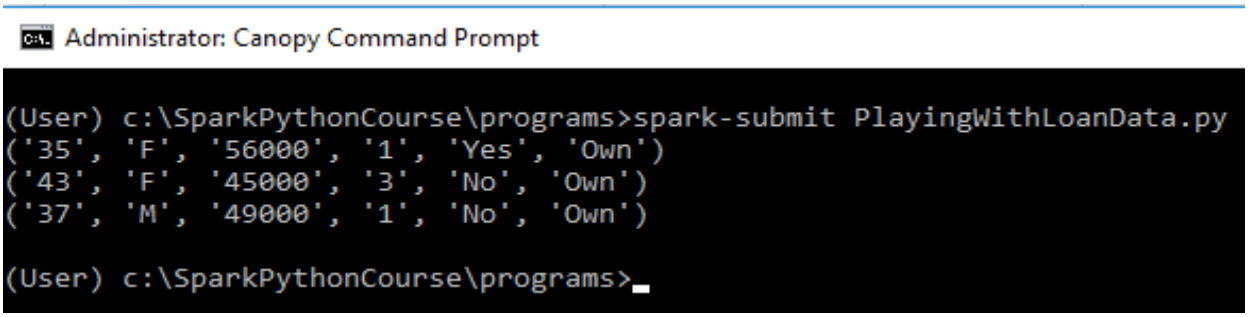
PlayingWithLoanData.py

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("LoanDataPractice")
sc = SparkContext(conf = conf)

lines = sc.textFile("file:///SparkPythonCourse/data/PracticeLoanData.txt")

def ParsingLines(lines):
    split2fields = lines.split()
    age = split2fields[0]
    sex = split2fields[1]
    hhincome = split2fields[2]
    numdep = split2fields[3]
    carloan = split2fields[4]
    homeownership = split2fields[5]
    return (age, sex, hhincome, numdep, carloan, homeownership)

ParsedLines = lines.map(ParsingLines)
OwnHome = ParsedLines.filter(lambda x: "Own" in x[5])
result = OwnHome.collect()
print(result)
```

Administrator: Canopy Command Prompt

```
(User) c:\SparkPythonCourse\programs>spark-submit PlayingWithLoanData.py
[('35', 'F', '56000', '1', 'Yes', 'Own'), ('43', 'F', '45000', '3', 'No', 'Own'), ('37', 'M', '49000', '1', 'No', 'Own')
]

(User) c:\SparkPythonCourse\programs>
```

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("LoanDataPractice")
sc = SparkContext(conf = conf)

lines = sc.textFile("file:///SparkPythonCourse/data/PracticeLoanData.txt")

def ParsingLines(lines):
    split2fields = lines.split()
    age = split2fields[0]
    sex = split2fields[1]
    hhincome = split2fields[2]
    numdep = split2fields[3]
    carloan = split2fields[4]
    homeownership = split2fields[5]
    return (age, sex, hhincome, numdep, carloan, homeownership)

ParsedLines = lines.map(ParsingLines)
OwnHome = ParsedLines.filter(lambda x: "Own" in x[5])
result = OwnHome.collect()

for resultant in result:
    print(resultant)
```



Distinct is used to find distinct type of categories involved in data. This can produce RDD with distinct categories or values. As can be seen in the script and result snippets, we can find the distinct type of home ownership from the above practice dataset.

PlayingWithLoanDataDistinct.py

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("LoanDataPractice")
sc = SparkContext(conf = conf)

lines = sc.textFile("file:///SparkPythonCourse/data/PracticeLoanData.txt")

def ParsingLines(lines):
    split2fields = lines.split()
    age = split2fields[0]
    sex = split2fields[1]
    hhincome = split2fields[2]
    numdep = split2fields[3]
    carloan = split2fields[4]
    homeownership = split2fields[5]
    return (age, sex, hhincome, numdep, carloan, homeownership)

ParsedLines = lines.map(ParsingLines)
OwnershipType = ParsedLines.map(lambda x:  x[5] )
DistinctOwnerType = OwnershipType.distinct().collect()
NumberofTypes = OwnershipType.distinct().count()

print('Number of Ownership Types: ',  NumberofTypes)

for ownertype in DistinctOwnerType:
    print(ownertype)
```

Administrator: Canopy Command Prompt

```
(User) c:\SparkPythonCourse\programs>spark-submit PlayingWithLoanDataDistinct.py
Number of Ownership Types:  2
Own
Rent

(User) c:\SparkPythonCourse\programs>
```

Union of RDDs can be done by simply using the union command but that depends on the circumstances. If there are many RDDs or RDDs generated over a number of executions and union of RDDs is required for computational purposes, we may define a function to make union for a large number of RDDs. In the following example we apply mapping and filtering to construct two distinct RDDs and then apply union to get back the original RDD.

From SparkContext we may define two RDDs like

Rdd1 = sc.parallelize([1,2])
Rdd2 = sc.parallelize([3,4])
Rdd1unionRdd2 = sc.union([Rdd1, Rdd2])
Rdd1unionRdd2.collect() provides the following output
[1,2,3,4]

Another approach is to use RDD1.union(RDD2) as can be seen in the following snippets.

PlayingWithLoanDataUnion.py

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("LoanDataPractice")
sc = SparkContext(conf = conf)

lines = sc.textFile("file:///SparkPythonCourse/data/PracticeLoanData.txt")

def ParsingLines(lines):
    split2fields = lines.split()
    age = split2fields[0]
    sex = split2fields[1]
    hhincome = split2fields[2]
    numdep = split2fields[3]
    carloan = split2fields[4]
    homeownership = split2fields[5]
    groupid = split2fields[6]
    return (age, sex, hhincome, numdep, carloan, homeownership, groupid) # included group id

ParsedLines = lines.map(ParsingLines) # creates RDD by mapping
group1rdd = ParsedLines.filter(lambda x: "1" in  x[6] ) # creates RDD of group ID
group2rdd = ParsedLines.filter(lambda x: "2" in  x[6] ) # creates RDD of group ID

print("customers of group 1:")
for line in group1rdd.collect():
    print(line)

print("customers of group 2:")
for line in group2rdd.collect():
    print(line)

RDDunion = group1rdd.union(group2rdd)
print("customers from union of group 1 and group 2:")
for line in RDDunion.collect():
    print(line)
```

```
(User) c:\SparkPythonCourse\programs>spark-submit PlayingWithLoanDataUnion.py
customers of group 1:
('35', 'F', '56000', '1', 'Yes', 'Own', '1')
('53', 'M', '65000', '1', 'Yes', 'Rent', '1')
customers of group 2:
('44', 'M', '57000', '0', 'No', 'Rent', '2')
('39', 'F', '76000', '2', 'Yes', 'Rent', '2')
('43', 'F', '45000', '3', 'No', 'Own', '2')
('37', 'M', '49000', '1', 'No', 'Own', '2')
customers from union of group 1 and group 2:
('35', 'F', '56000', '1', 'Yes', 'Own', '1')
('53', 'M', '65000', '1', 'Yes', 'Rent', '1')
('44', 'M', '57000', '0', 'No', 'Rent', '2')
('39', 'F', '76000', '2', 'Yes', 'Rent', '2')
('43', 'F', '45000', '3', 'No', 'Own', '2')
('37', 'M', '49000', '1', 'No', 'Own', '2')
```

Addition is used to add one RDD with the other (that essentially makes a union of RDDs). For example

RDD1 = sc.parallelize([1,2,3])
RDD2 = sc.parallelize([2,3,4])
(RDD1 + RDD2).collect() provides the following output
[1,2,3,2,3,4]

```
RDDunion = group1rdd + group2rdd
print("customers from union of group 1 and group 2:")
for line in RDDunion.collect():
    print(line)
```

Using addition (by using +) in the above snippet will provide following outcome.

```
customers from union of group 1 and group 2:
('35', 'F', '56000', '1', 'Yes', 'Own', '1')
('53', 'M', '65000', '1', 'Yes', 'Rent', '1')
('44', 'M', '57000', '0', 'No', 'Rent', '2')
('39', 'F', '76000', '2', 'Yes', 'Rent', '2')
('43', 'F', '45000', '3', 'No', 'Own', '2')
('37', 'M', '49000', '1', 'No', 'Own', '2')
```

Intersection between RDDs can be done as follows

```
RDDunion = group1rdd.union(group2rdd)
RDDIntersection = RDDunion.intersection(group2rdd)
print("customers from intersection of RDDunion and group2rdd is group2rdd:")
for line in RDDIntersection.collect():
    print(line)
```

```
customers from intersection of RDDunion and group2rdd is group2rdd:
('37', 'M', '49000', '1', 'No', 'Own', '2')
('43', 'F', '45000', '3', 'No', 'Own', '2')
('39', 'F', '76000', '2', 'Yes', 'Rent', '2')
('44', 'M', '57000', '0', 'No', 'Rent', '2')
```

4. Descriptive Statistics (Actions on RDD)

There are some common operations and actions that produce summary statistics from Spark Context RDD. Some of such very commonly used functions are discussed below:

Sum: Let us consider an RDD with values 1, 2, 3 defined as sc.parallelize([1, 2, 3]) and the sum can be obtained by using an action for that RDD. The following action on RDD produces the result 6.

```
sc.parallelize([1, 2, 3]).sum()
```

Mean: This instance produces the mean from an RDD, where the following command produces (1+2+3)/3 = 2.0

```
sc.parallelize([1, 2, 3]).mean()
```

Variance: This produces the variance

```
sc.parallelize([1, 2, 3]).variance()
```

Sample Variance: This produces sample variance, replacing the N by N-1

```
sc.parallelize([1, 2, 3]).sampleVariance()
```

Standard Deviation: To obtain standard deviation use

```
sc.parallelize([1, 2, 3]).stdev()
```

Sample Standard Deviation: Replaces N by N-1 in computation of standard deviation.

```
sc.parallelize([1, 2, 3]).sampleStdev()
```

Minimum and Maximum: These values can be obtained by using the following commands

```
sc.parallelize([4, 2, 5, 1, 2, 5, 1, 1]).takeOrdered(1) # gets minimum

sc.parallelize([4, 2, 5, 1, 2, 5, 1, 1]).takeOrdered(1,key= lambda x: -x)
```

ActionForStatisticsOnRDD.py

```
1  from pyspark import SparkConf, SparkContext
2
3  conf = SparkConf().setMaster("local").setAppName("LoanDataPractice")
4  sc = SparkContext(conf = conf)
5
6  rdd1 = sc.parallelize([1, 2, 3])
7
8  print("Some of Values in RDD1: ", rdd1.sum() )
9  print("Variance of Values in RDD1: ", rdd1.variance() )
10 print("Standard Deviation of Values in RDD1: ", rdd1.stdev() )
11 print("Sample Variance of Values in RDD1: ", rdd1.sampleVariance() )
12 print("Sample Standard Deviation of Values in RDD1: ", rdd1.sampleStdev() )
13
14 print("Variance of Values in RDD1 Upto 2 Decimal Places: ", round( rdd1.variance(),2) )
15
16 rdd2 = sc.parallelize([4,2,5,1,2,5,1,1])
17 print( "Minimum Value in RDD2: ", rdd2.takeOrdered(1) )
18 print( "Lowest 2 Values in RDD2: ", rdd2.takeOrdered(2) )
19 print( "Lowest Distinct 2 Values in RDD2: ", rdd2.distinct().takeOrdered(2) )
20 print("Maximum Value in RDD2: ", rdd2.takeOrdered(1, key=lambda x:-x))
21 print( "Highest 2 Values in RDD2: ", rdd2.takeOrdered(2, key=lambda x: -x) )
22 print( "Highest Distinct 2 Values in RDD2: ", rdd2.distinct().takeOrdered(2, key=lambda x: -x) )
23 print( "Ascending Ordered Values in RDD2: ", rdd2.takeOrdered( rdd2.count() ))
24 print("Descending Ordered Values in RDD2: ", rdd2.takeOrdered( rdd2.count(), key=lambda x: -x)  )
```

```
(User) c:\SparkPythonCourse\programs>spark-submit ActionForStatisticsOnRDD.py
Some of Values in RDD1:  6
Variance of Values in RDD1:  0.6666666666666666
Standard Deviation of Values in RDD1:  0.816496580928
Sample Variance of Values in RDD1:  1.0
Sample Standard Deviation of Values in RDD1:  1.0
Variance of Values in RDD1 Upto 2 Decimal Places:  0.67
Minimum Value in RDD2:  [1]
Lowest 2 Values in RDD2:  [1, 1]
Lowest Distinct 2 Values in RDD2:  [1, 2]
Maximum Value in RDD2:  [5]
Highest 2 Values in RDD2:  [5, 5]
Highest Distinct 2 Values in RDD2:  [5, 4]
Ascending Ordered Values in RDD2:  [1, 1, 1, 2, 2, 4, 5, 5]
Descending Ordered Values in RDD2:  [5, 5, 4, 2, 2, 1, 1, 1]
```

5. Key Value Pair and RDDs

Count By Key: The following command produces a dictionary by counting key value pairs, for "a" we have 2 instances of pairs, and for "b" we have one instance for value pair.

```
rdd1 = sc.parallelize([("a", 1), ("b", 2), ("a",2)])
print( rdd1.countByKey().items() )
```

Group By Key: This is used to group values by corresponding keys. This groups the values corresponding to a key in a single sequence.

```
rdd1 = sc.parallelize([("a", 1), ("b", 2), ("a",2)])
print( rdd1.groupByKey().collect() ) # produces result iterable object
print( rdd1.groupByKey().mapValues(list).collect() )
```

Subtract By Key: Keys in RDD is subtracted, and is not subtracted in the absence of a pair.

```
rdd4 = sc.parallelize([("a", 1), ("a", 1), ("b", 5), ("a", 2)])
rdd5 = sc.parallelize([("a", 3), ("c", None), ("b",2)])
print("Subtract by key, both keys are subtracted resulting []: ",
sorted(rdd4.subtractByKey(rdd5).collect()) )
# the above commands produces []

rdd6 = sc.parallelize([("a", 3), ("c", 1)])
print("Subtract by key, key a is subtracted but not key b: ",
sorted(rdd4.subtractByKey(rdd6).collect()) )
# the above commands produce [("b", 5)]
```

Reduce By Key: Generally, a lambda function is used to provide a reduce option to reduce by key

```
rdd4.reduceByKey( lambda x, y: x + y).collect()
# this produces [(“a”, 4), (“b”, 5)]
```

KeyValuePairRDD.py

```
2
3 conf = SparkConf().setMaster("local").setAppName("LoanDataPractice")
4 sc = SparkContext(conf = conf)
5
6 rdd1 = sc.parallelize([("b", 1), ("a", 2), ("b",2), ("a", 1), ("a",3)])
7
8 # Count by key
9
10 print( "Count By Key Pairs in RDD1: ", rdd1.countByKey().items() )
11 print( "Count By Key Pairs and Sort By Keys in RDD1: ", sorted( rdd1.countByKey().items() ) )
12
13 # Group by key
14
15 rdd2 = rdd1.groupByKey()
16 rdd3 = rdd2.map( lambda x: (x[0], list(x[1]))).collect()
17 print("Group by key, values in list: ", rdd3)
18
19 print( "Group by key, values in list (one liner and simpler than above): ", rdd1.groupByKey().mapValues(list).collect() )
20
21 # Subtract by key
22
23 rdd4 = sc.parallelize([("a", 1), ("a", 1), ("b", 5), ("a", 2)])
24 rdd5 = sc.parallelize([("a", 3), ("c", None), ("b",2)])
25 print("Subtract by key, both keys are subtracted resulting []: ",  sorted(rdd4.subtractByKey(rdd5).collect()) )
26
27 rdd6 = sc.parallelize([("a", 3), ("c", 1)])
28 print("Subtract by key, key a is subtracted but not key b: ", sorted(rdd4.subtractByKey(rdd6).collect()) )
29
30 # Reduce by key
31
32 print( "Reduce by key: forming addition by keys: ", rdd4.reduceByKey( lambda x, y: x + y).collect() )
```

```
(User) c:\SparkPythonCourse\programs>spark-submit KeyValuePairRDD.py
Count By Key Pairs in RDD1:  dict_items([('a', 3), ('b', 2)])
Count By Key Pairs and Sort By Keys in RDD1:  [('a', 3), ('b', 2)]
Group by key, values in list:  [('a', [2, 1, 3]), ('b', [1, 2])]
Group by key, values in list (one liner and simpler than above):  [('a', [2, 1, 3]), ('b', [1, 2])]
Subtract by key, both keys are subtracted resulting []:  []
Subtract by key, key a is subtracted but not key b:  [('b', 5)]
Reduce by key: forming addition by keys:  [('a', 4), ('b', 5)]
```

6. Joining RDDs

Left and Right Outer Join: This action performs a left outer join of one RDD with the other.

```
rdd1 = sc.parallelize([("a", 1), ("b", 4)])
rdd2 = sc.parallelize([("a", 2), ("c", 3)])
print("Left outer join: ", sorted(rdd1.leftOuterJoin(rdd2).collect()) )
            # this produces [ ("a", (1,2)), ("b", (4, None)) ]

print( "Left outer join: ", sorted(rdd2.leftOuterJoin(rdd1).collect()) )
            # this produces [ ("a", (2,1)), ("c", (3, None)) ]

print( "Right outer join: ", sorted(rdd1.rightOuterJoin(rdd2).collect()) )
            # this produces [ ("a", (1,2)), ("c", (None, 3)) ]

print( "Right outer join: ", sorted(rdd2.rightOuterJoin(rdd1).collect()) )
            # this produces [ ("a", (2,1)), ("b", (None, 4)) ]
```

```
Left outer join:  [('a', (1, 2)), ('b', (4, None))]
Left outer join:  [('a', (2, 1)), ('c', (3, None))]
Right outer join:  [('a', (1, 2)), ('c', (None, 3))]
Right outer join:  [('a', (2, 1)), ('b', (None, 4))]
```

6. Data Analysis (Producing Summary Statistics)

6.1 Configuration

Open a Jupyter Ipython Notebook and do necessary configuration (shown below and in Chapter 1).

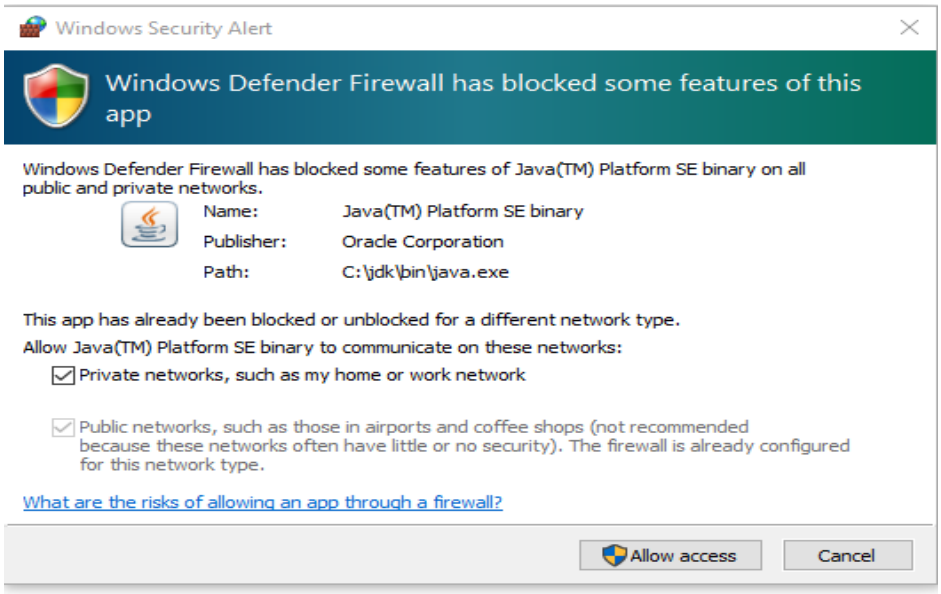
```
In [1]: !pip install findspark
```

```
Requirement already satisfied: findspark in c:\anaconda3\lib\site-packages
```

```
You are using pip version 9.0.1, however version 10.0.1 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

```
In [2]: import findspark
findspark.init()
import pyspark
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("EnergyRating")
sc = SparkContext(conf = conf)
```

If a pop-up window like below occurs, please Allow Access, otherwise will be blocked by the windows firewall system.



6.2 Calling Data File and Working on RDDs

Next, we call a file (PracticeLoanData.txt) and define a function (ParsingLines), and finally use them for operations and actions on RDDs. The following snippet will then produce outputs.

```
lines = sc.textFile("file:///F:/SparkPythonCourse/data/PracticeLoanData.txt")
def ParsingLines(lines):
    split2fields = lines.split()
    age = split2fields[0]
    sex = split2fields[1]
    hhincome = split2fields[2]
    numdep = split2fields[3]
    carloan = split2fields[4]
    homeownership = split2fields[5]
    groupid = split2fields[6]
    return (age, sex, hhincome, numdep, carloan, homeownership, groupid) # included group id

ParsedLines = lines.map(ParsingLines) # creates RDD by mapping
group1rdd = ParsedLines.filter(lambda x: "1" in  x[6] ) # creates RDD of group ID
group2rdd = ParsedLines.filter(lambda x: "2" in  x[6] ) # creates RDD of group ID

print("customers of group 1:")
for line in group1rdd.collect():
    print(line)

print("customers of group 2:")
for line in group2rdd.collect():
    print(line)

RDDunion = group1rdd.union(group2rdd)
print("customers from union of group 1 and group 2:")
for line in RDDunion.collect():
    print(line)
```

Outputs obtained after running the above snippet would look like this.

```
customers of group 1:
('35', 'F', '56000', '1', 'Yes', 'Own', '1')
('53', 'M', '65000', '1', 'Yes', 'Rent', '1')
customers of group 2:
('44', 'M', '57000', '0', 'No', 'Rent', '2')
('39', 'F', '76000', '2', 'Yes', 'Rent', '2')
('43', 'F', '45000', '3', 'No', 'Own', '2')
('37', 'M', '49000', '1', 'No', 'Own', '2')
customers from intersection of RDDunion and group2rdd is group2rdd:
('43', 'F', '45000', '3', 'No', 'Own', '2')
('37', 'M', '49000', '1', 'No', 'Own', '2')
('39', 'F', '76000', '2', 'Yes', 'Rent', '2')
('44', 'M', '57000', '0', 'No', 'Rent', '2')
```

6.4 Construction of Data Frame

For example, we have an RDD from group 1, that is, group1rdd. Now, we want to convert this RDD into a data frame for easy manipulation of data for further use. We can use the following code snippet to get a data frame from existing RDD.

```
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)
group1dframe = sqlContext.createDataFrame(group1rdd)
group1dframe.show()
```

```
+---+---+-----+---+---+----+---+
| _1| _2|   _3| _4| _5|  _6| _7|
+---+---+-----+---+---+----+---+
| 35|  F|56000|  1|Yes| Own|  1|
| 53|  M|65000|  1|Yes|Rent|  1|
+---+---+-----+---+---+----+---+
```

```
group1df = group1dframe.selectExpr("_1 as age", "_2 as sex", "_3 as hhincome",
          "_4 as numdep", "_5 as carloan", "_6 as homeownership", "_7 as groupid")
group1df.show()
group1df.printSchema()
```

```
+---+---+--------+------+-------+-------------+-------+
|age|sex|hhincome|numdep|carloan|homeownership|groupid|
+---+---+--------+------+-------+-------------+-------+
| 35|  F|   56000|     1|    Yes|          Own|      1|
| 53|  M|   65000|     1|    Yes|         Rent|      1|
+---+---+--------+------+-------+-------------+-------+
```

6.5 Summary Statistics from Data Frames

A very simple command dataframe.describe ('column name').show() will provide you with summary statistics such as number of counts, mean, standard deviation, minimum and maximum.

```
group1df.describe('age', 'hhincome').show()
```

```
+-------+------------------+-----------------+
|summary|               age|         hhincome|
+-------+------------------+-----------------+
|  count|                 2|                2|
|   mean|              44.0|          60500.0|
| stddev|12.727922061357855|6363.961030678927|
|    min|                35|            56000|
|    max|                53|            65000|
+-------+------------------+-----------------+
```

If you want to calculate a single satistic, say for exaple, mean age, then use the command

```
from pyspark.sql.functions import mean, min, max
group1df.select([mean('age')]).show()
```

```
+--------+
|avg(age)|
+--------+
|    44.0|
+--------+
```

For cross tabulation of two variables sex and homeownership follow the snippet below:

```
group1df.stat.crosstab("sex", "homeownership").show()
```

```
+-----------------+---+----+
|sex_homeownership|Own|Rent|
+-----------------+---+----+
|                M|  0|   1|
|                F|  1|   0|
+-----------------+---+----+
```

In a nutshell, we have learnt how to play with the data for different operations and actions on RDD. Finally, we have constructed data frame and made that data frame ready for statistical analysis. Our tiny piece of data set in data frame form looks like the one produced below.

```
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)
group1and2dframe = sqlContext.createDataFrame(RDDunion)
group12df = group1and2dframe.selectExpr("_1 as age", "_2 as sex", "_3 as hhincome",
        "_4 as numdep", "_5 as carloan", "_6 as homeownership", "_7 as groupid")
group12df.show()
#group12df.printSchema()
```

```
+---+---+--------+------+-------+-------------+-------+
|age|sex|hhincome|numdep|carloan|homeownership|groupid|
+---+---+--------+------+-------+-------------+-------+
| 35|  F|   56000|     1|    Yes|          Own|      1|
| 53|  M|   65000|     1|    Yes|         Rent|      1|
| 44|  M|   57000|     0|     No|         Rent|      2|
| 39|  F|   76000|     2|    Yes|         Rent|      2|
| 43|  F|   45000|     3|     No|          Own|      2|
| 37|  M|   49000|     1|     No|          Own|      2|
+---+---+--------+------+-------+-------------+-------+
```

```
group12df.stat.crosstab("sex", "homeownership").show()
```

```
+-----------------+---+----+
|sex_homeownership|Own|Rent|
+-----------------+---+----+
|                M|  1|   2|
|                F|  2|   1|
+-----------------+---+----+
```

Cool, now go ahead with further analysis!!!