The Python code in this tutorial can be run in

1. Databricks (Cloud based)
   - Log in to your Databricks account and start a new notebook.
   - Type in and run the code in the notebook.

2. Ubuntu VM (Local setup)
   - Start PySpark shell

     $ pyspark

   - Type in and run the code in the PySpark shell.
   - Alternatively, you may also use Jupyter Notebook to run the code if you have installed Anaconda on the Ubuntu VM.

3. Google Colab 

   You need to do Apache Spark installation and setup in Google Colab. Here are some reference links:

   - Running Pyspark in Colab
   - PySpark Google Colab | Working With PySpark in Colab
   - Getting Started Spark 3.0.0 in Google Colab

1. Creating PySpark session

- SparkContext is an entry point to interact with underlying Spark functionality.
- SparkContext can be accessed in the PySpark as a variable named *sc*.

```
### Create a Spark session on a Spark cluster

# Import the PySpark module
from pyspark.sql import SparkSession

# Create SparkSession object

spark = SparkSession.builder \
                    .master('local[*]') \
                    .appName('test') \
                    .getOrCreate()

# Show version of Spark
print(spark.version)

# Get the spark context
sc=spark.sparkContext
print('Spark version: ', sc.version)

# Show the version of Python used by Spark
print('Python version for Spark: ', sc.pythonVer)

# Master is the URL of the cluster or “local” string to run in local mode.
print('Master of Spark Context: ', sc.master)

3.1.0
Spark version:  3.1.0
Python version for Spark:  3.8
Master of Spark Context:  local[8]
```

2. Programming in PySpark RDD’s

2.1 Lambda function

Lambda functions are anonymous functions (without a function name). A lambda function can have any number of arguments but only one expression and the expression is evaluated and returned (no return statement). The general form of lambda function:

lambda arguments: expression

It is very powerful and often used in conjunction with typical functional concepts like *map()* and *filter()* functions.

- **Map()** function takes a function and an iterable (list, tuple, etc), and return a new list after applying the given function to each item of the given iterable. The general syntax of map function: *map(function, list)*
  - A lambda functions can be used with map(). The general syntax of map() function with lambda() is *map(lambda :, iter)*.
- **Filter()** function takes a function and a list and return a new list for which the function evaluates as true. The general syntax of filter function: *filter(function, list)*
  - A lambda function can be used with filter(). The general syntax of the filter() function with lambda() is *filter(lambda :, list)*.

```
### Use of lambda() with map()

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Print my_list in the console
print("Input list is", my_list)

# Square all numbers in my_list
squared_list_lambda = list(map(lambda x: x**2, my_list))

# Print the result of the map function
print("The squared numbers are", squared_list_lambda)


### Use of lambda() with filter()
my_list2 = [10, 21, 31, 40, 51, 60, 72, 80, 93, 101]
# Print my_list2 in the console
print("Input list is:", my_list2)

# Filter numbers divisible by 10
filtered_list = list(filter(lambda x: (x%10 == 0), my_list2))

# Print the numbers divisible by 10
print("Numbers divisible by 10 are:", filtered_list)

Input list is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The squared numbers are [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Input list is: [10, 21, 31, 40, 51, 60, 72, 80, 93, 101]
Numbers divisible by 10 are: [10, 40, 60, 80]
```

2.2 Abstracting Data with RDDs

RDD stands for Resilient Distributed Datasets. It is simply a collection of data distributed across the cluster. RDD is the fundamental and backbone data type in PySpark. When Spark starts processing data, it divides the data into partitions and distributes the data across cluster nodes, with each node containing a slice of data.

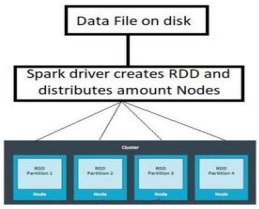
Important properties of a RDD:

- Resilient means the ability to withstand failures and recompute missing or damaged partitions.
- Distributed means spanning the jobs across multiple nodes in the cluster for efficient computation.
- Datasets is a collection of partitioned data e.g. Arrays, Tables, Tuples or other objects.

Methods to create RDDs:

- The simplest method to create RDDs is to take an existing collection of objects (eg. a list, an array or a set) and pass it to SparkContext's parallelize method, As a result, a parallelized collection (parallelizing) is created
- A more common way to create RDDs is to load data from external datasets such as files stored in HDFS or objects in Amazon S3 buckets or from lines in a text file stored locally and pass it to SparkContext's textFile method.
- Finally, RDDs can also be created from existing RDDs

A partition in Spark is the logical division of the large dataset with each part being stored in multiple locations across the cluster. By default Spark partitions the data at the time of creating RDD based on several factors such as available resources, external datasets etc, however, this behavior can be controlled by passing a second argument called 'minPartitions' which defines the minimum number of partitions to be created for an RDD. The number of partitions in an RDD can always be found by using the 'getNumPartitions' method.




```
### 1. Create RDD from a collection and turn it into parallelized collection
# Use parallelize method for parallelizing an existing collection of objects (e.g. list, array, etc).

# Create an RDD from a list of words
RDD = sc.parallelize(["Spark", "is", "a", "framework", "for", "Big Data processing"])

# Print out the type of the created object
print("The type of RDD is", type(RDD))
```

```
### 2. Create RDD from external dataset
# Use textFile metod to load data from external datasets (e.g. files in HDFS, Amazon S3 objects, local text files).
# This the most common method to create RDDs

# Make sure the data file has been uploaded to the Databricks file system
file_path='dbfs:/FileStore/shared_uploads/wei.jie@uwl.ac.uk/README.md'
# Print the file_path
print("The file_path is", file_path)

# Create a fileRDD from file_path
fileRDD = sc.textFile(file_path)

# Check the type of fileRDD
print("The file type of fileRDD is", type(fileRDD))

# Check the number of partitions in fileRDD
print("Number of partitions in fileRDD is", fileRDD.getNumPartitions())

# Create a fileRDD_part from file_path with 5 partitions
fileRDD_part = sc.textFile(file_path, minPartitions = 5)

# Check the number of partitions in fileRDD_part
print("Number of partitions in fileRDD_part is", fileRDD_part.getNumPartitions())
```

```
The type of RDD is <class 'pyspark.rdd.RDD'>
The file_path is dbfs:/FileStore/shared_uploads/wei.jie@uwl.ac.uk/README.md
The file type of fileRDD is <class 'pyspark.rdd.RDD'>
Number of partitions in fileRDD is 2
Number of partitions in fileRDD_part is 5
```

2.3 Basic RDD Transformations and Actions

RDD in PySpark supports two different types of operations:

- **Transformations** are operations on RDDs that return a new RDD;
- **Actions** are operations that perform some computation on the RDD.

Spark creates a graph from all the operations you perform on an RDD and execution of the graph starts only when an action is performed on RDD as shown in this figure. This is called lazy evaluation in Spark.

RDD Transformations:

- **map()** transformation 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.
- **filter()** takes in a function and returns an RDD that only has elements that pass the condition. It is useful for filtering large datasets based on a keyword.
- **flatMap()** is similar to map transformation except it returns multiple values for each element in the source RDD. A simple usage of flatMap is splitting an input string into words.
- **union()** returns the union of one RDD with another RDD.

RDD Actions: Actions are the operations that are applied on RDDs to return a value after running a computation. The four basic actions that you'll learn in this lesson are collect, take, first and count.

- **collect()** is an action operation that returns complete list of elements from a RDD, and it should only be used to retrieve results for small datasets.
- **count()** returns the number of elements in the RDD, while take(N) returns an array with the first N element of the dataset.
- **take(n)** returns an array with the first n elements of the dataset.
- **first()** returns the first element of the dataset (similar to take(1)).

```
### 1. Map and collection on RDD

numbRDD =  sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Create map() transformation to cube numbers
cubedRDD = numbRDD.map(lambda x: x ** 3)

# Collect the results
numbers_all = cubedRDD.collect()

# Print the numbers from numbers_all
for numb in numbers_all:
    print(numb)

### 2. Filter and count on RDD
# Make sure the data file has been uploaded to the Databricks file system
file_path='dbfs:/FileStore/shared_uploads/wei.jie@uwl.ac.uk/Spark.md'
# Create a fileRDD from file_path
fileRDD = sc.textFile(file_path)

# Create filter() transformation to select the lines containing the keyword Spark
fileRDD_filter = fileRDD.filter(lambda line: 'Spark' in line)

# How many lines are there in fileRDD?
print("\n The total number of lines with the keyword Spark is", fileRDD_filter.count())

### 3. First and take on RDD
# Print the first line filtered fileRDD
fileRDD_filter.first()

# Print the first 3 lines of filtered fileRDD
for line in fileRDD_filter.take(3):
    print(line)

### 4. flatMap on RDD
# split the lines of the filtered fileRDD into individual words
fileRDD_flatMap = fileRDD_filter.flatMap(lambda x: x.split(' '))

# Print the first 10 words of the
for word in fileRDD_flatMap.take(10):
    print(word)
```

```
1
8
27
64
125
216
343
512
729
1000

 The total number of lines with the keyword Spark is 7
 'Examples for Learning Spark',
 'Examples for the Learning Spark book. These examples require a number of libraries and as such have long build files. We have also added a stand alone example wit
h minimal dependencies and a small build file',
 'These examples have been updated to run against Spark 1.3 so they may',

'Examples
for
```

2.4 Pair RDDs in PySpark

Pair RDD in PySpark is a special data structure to work with key/value dataset. In pair RDDs, the key refers to the identifier, whereas value refers to the data.

The two most common ways of creating pair RDD:

- Creating from a list of the key-value tuple
- Creating from a regular RDD.

All regular RDD transformations work on pair RDD. Since pair RDDs contain tuples, special functions are needed to operate on key-value pairs, such as reduceByKey, groupByKey, sortByKey and join.

- **reduceByKey()** transformation combines values with the same key using an associative reduce function. It runs parallel operations for each key in the dataset. It returns a new RDD consisting of each key and the reduced value for that key.
- **sortByKey()** transformation operation returns an RDD sorted by key in ascending or descending order.
- **groupByKey()** transformation operation groups all the values with the same key in the pair RDD.
- **join()** transformation merges two pair RDDs together based on their key.

```
### 1. ReduceBykey and Collect
# reduceByKey() operates on key, value (k,v) pairs and
# merges the values for each key

# Create PairRDD Rdd with key value pairs (player, goal)
Rdd = sc.parallelize([('Messi', 23), ('Ronaldo', 34), ('Ronaldo', 25), ('Neymar', 26), ('Messi',24)])

# Transform the RDD with reduceByKey() into a pair RDD 'Rdd_Reduced'
# by adding the values with the same key.
Rdd_Reduced = Rdd.reduceByKey(lambda x, y: x + y)

# Iterate over the result and print the output
for num in Rdd_Reduced.collect():
  print("Key {} has {} Counts".format(num[0], num[1]))


### 2. SortByKey and Collect
# sortByKey() sorts the pair RDD based on the key

# Sort the reduced RDD with the key (i.e. player name) by descending order
Rdd_Reduced_Sort = Rdd_Reduced.sortByKey(ascending=False)
# Iterate over the result and print the output
print('\nSort by player name by descending order: ')
for kv in Rdd_Reduced_Sort.collect():
  print('{} has {} goals'.format(kv[0], kv[1]))

# Sort the reduced RDD with number of goals by ascending order
Rdd_Reduced_Rev = Rdd_Reduced.map(lambda x: (x[1], x[0]))    # swap key with value
print("\nSort by number of goals by ascending order:")
for kv in Rdd_Reduced_Rev.sortByKey(ascending=True).collect():    # sort by number of goals ascendingly
    print('{} has {} goals'.format(kv[1], kv[0]))


### 3. GroupByKey and Join
# group the airports for the same country with groupByKey() transformation
airports = [('US', 'JFK'), ('UK', 'LHR'), ('UK', 'MAN'), ('US', 'SFO'), ('UK', 'LGW'), ('FR', 'CDG')]
Rdd = sc.parallelize(airports)
Rdd_group = Rdd.groupByKey().collect()
print('\nGroup airports in each country:')
for country, airport in Rdd_group:
    print(country, list(airport))

# Merge two pair RDDs together based on their key with join() transformation
Rdd1 = sc.parallelize([('Messi', 34), ('Ronaldo', 32), ('Neymar', 24)])
Rdd2= sc.parallelize([('Ronaldo', 80), ('Neymar', 50), ('Messi', 80)])
print('\nMerge RDDs:')
Rdd1.join(Rdd2).collect()
```

```
Key Ronaldo has 59 Counts
Key Neymar has 26 Counts
Key Messi has 47 Counts

Sort by player name by descending order:
Ronaldo has 59 goals
Neymar has 26 goals
Messi has 47 goals

Sort by number of goals by ascending order:
Neymar has 26 goals
Messi has 47 goals
Ronaldo has 59 goals

Group airports in each country:
FR ['CDG']
UK ['LHR', 'MAN', 'LGW']
US ['JFK', 'SFO']

Merge RDDs:
Out[5]: [('Neymar', (24, 50)), ('Ronaldo', (32, 80)), ('Messi', (34, 80))]
```

2.5 Advanced RDD Actions

Some advanced actions on RDDs:

- **reduce(func)** action is used for aggregating the elements of a regular RDD. The taking-in function should be commutative (changing the order of the operands does not changed the result) and associative so that it can be computed correctly in parallel.
- **saveAsTextFile()** action saves RDD into a text file inside a directory with each partition as a separate file. Using *coalesce()* method can save RDD as a single text file.
- **countByKey()** action is only applicable to pair RDD, which is used to count the number of elements for each key.
- **collectAsMap()** action return the key-value pairs in the RDD as a dictionary

```
### 1. CountByKey
# Keys for a pair RDD can be counted by countByKey() action.

# count the number of elements for each key
Rdd = sc.parallelize([('a', 1), ('b', 1), ('a', 2), ('a', 3)])
for kee, val in Rdd.countByKey().items():
                print(kee, ':', val)


### 2. Reduce
# use reduce() to compute the sum of all elements in a RDD
x = [1, 3, 5 , 7]
RDD = sc.parallelize(x)
sum = RDD.reduce(lambda x, y: x+y)
print('Result of reduce: ', sum)


### 3. SaveAsTextFile
# x = ['Sam', 'Peter', 'Alice', 'Tom']
# RDD = sc.parallelize(x)
# RDD.saveAsTextFile('tempFile1')        # check your current working directory and see if a new directory created
# RDD.coalesce(1).saveAsTextFile('tempFile2')          # Save as a single file


### 4. CollectAsMap
dict= sc.parallelize([(1,2), (3, 4)]).collectAsMap()
print('Type of result: ', type(dict))
print('Result of collectAsMap: ', dict)
```

```
a : 3
b : 1
Result of reduce:  16
Type of result:  <class 'dict'>
Result of collectAsMap:  {1: 2, 3: 4}
```