***What is PySpark?***

PySpark, released by Apache Spark community, is basically a Python API for supporting Python with Spark. By utilizing PySpark, you can work and integrate with RDD easily in Python. The library Py4j helps to achieve this feature.

There are several features of PySpark framework:

1. Faster processing than other frameworks.
2. Real-time computations and low latency due to in-memory processing.
3. Polyglot, which means compatible with several languages like Java, Python, Scala and R.
4. Powerful caching and efficient disk persistence.
5. Deployment can be performed by Hadoop through Yarn.

**Audience for PySpark Tutorial**

* The professionals who are aspiring to make a career in programming language and also those who want to perform real-time processing through framework can go for this PySpark tutorial.
* Also, those who want to learn PySpark along with its several modules, as well as submodules, must go for this PySpark tutorial.

## Prerequisites to PySpark

We assume that before learning PySpark, the readers already have basic knowledge about the programming language as well as frameworks. Also, it is recommended to have a sound knowledge of Spark, Hadoop, Scala Programming Language, HDFS as well as Python.

**Factors about PySpark API**

There are a few key differences between the Python and Scala APIs which we will discuss in this PySpark Tutorial:

* Since Python is dynamically typed, therefore PySpark RDDs can easily hold objects of multiple types.
* PySpark doesn’t support some API calls, like lookup and non-text input files. However, this feature will be added in future releases.

Although RDDs support the same methods as their Scala counterparts in PySpark but takes [**Python functions**](https://data-flair.training/blogs/python-function/) and returns Python collection types as a result. Using Python’s lambda syntax, short functions can be passed to RDD methods.

1. logData = sc.**textFile**(logFile).**cache**()
2. errors = logData.**filter**(lambda line: "ERROR" in line)

## Basically, the functions in PySpark which are defined with the def keyword;  can be passed easily. And, this is very beneficial for longer functions that cannot be shown using the lambda:

1. def **is\_error**(line):
2. return "ERROR" in line
3. errors = logData.**filter**(is\_error)

## Moreover, in enclosing scopes functions can access objects, however, to those objects modifications within RDD methods will not be propagated back:

## Also, to launch an interactive shell, PySpark fully supports interactive use, so simply run ./bin/pyspark.

## Installing and Configuring PySpark

Basically, we require Python 2.6 or higher for PySpark. Moreover, by using a standard CPython interpreter in order to support [**Python modules**](https://data-flair.training/blogs/python-modules/) that use C extensions, we can execute PySpark applications.

In addition, PySpark requires python to be available on the system PATH and use it to run programs by default. However, by setting the PYSPARK\_PYTHON environment variable in conf/spark-env.sh (or .cmd on Windows), an alternate Python executable may be specified.

Moreover, including Py4J, all of PySpark’s library dependencies, are in a bundle with PySpark.

Further, using the bin/pyspark script, Standalone PySpark applications must run. Also, using the settings in conf/spark-env.sh or .cmd, it automatically configures the Java as well as Python environment. And, while it comes to the bin/pyspark package, the script automatically adds to the PYTHONPATH.

## Interactive Use of PySpark

To run PySpark applications, the bin/pyspark script launches a Python interpreter. At first build Spark, then launch it directly from the command line without any options, to use [PySpark](https://pypi.org/project/pyspark/) interactively:

1. $ sbt/sbt assembly
2. $ ./bin/pyspark

## Also, to explore data interactively we can use the Python shell and moreover it is a simple way to learn the API:

1. words = sc.**textFile**("/usr/share/dict/words")
2. words.**filter**(lambda w: w.**startswith**("spar")).**take**(5)
3. [u'spar', u'sparable', u'sparada', u'sparadrap', u'sparagrass']
4. **help**(pyspark) # Show all pyspark functions

However, the bin/pyspark shell creates SparkContext that runs applications locally on a single core, by default. Further, set the MASTER environment variable, in order to connect to a non-local cluster, or also to use multiple cores.

**For example:**

If we want to use the bin/pyspark shell along with the standalone [**Spark cluster**](https://data-flair.training/blogs/apache-spark-cluster-managers-tutorial/):

1. $ MASTER=spark://IP:PORT ./bin/pyspark

## Or, to use four cores on the local machine:

1. $ MASTER=local[4] ./bin/pyspark

## IPython

Moreover, we can easily launch PySpark in IPython by following this PySpark tutorial. Here, IPython refers to an enhanced Python interpreter. Basically, with IPython 1.0.0, PySpark can easily work. Hence, set the IPYTHON variable to 1 at the time of running bin/pyspark, to use IPython:

1. $ IPYTHON=1 ./bin/pyspark

In addition, by setting IPYTHON\_OPTS, we can customize the of its command.

**For example:**

In order to launch the IPython Notebook by using PyLab graphing support:

1. $ IPYTHON\_OPTS="notebook --pylab inline" ./bin/pyspark

Moreover, if we set the MASTER environment variable, IPython also works on a cluster or on multiple cores.

## Standalone Programs

By creating a SparkContext in our script and by running the script using bin/pyspark, we can use PySpark from standalone **Python scripts**.

So, using the Python API (PySpark), we will see how to write a standalone application.

**For example**

Here we are creating a simple Spark application, SimpleApp1.py:

1. """SimpleApp1.py"""
2. from pyspark import SparkContext
3. logFile = "$YOUR\_SPARK\_HOME/README.md" # Should be some file on your system
4. sc = **SparkContext**("local", "Simple App1")
5. logData = sc.**textFile**(logFile).**cache**()
6. numAs = logData.**filter**(lambda s: 'a' in s).**count**()
7. numBs = logData.**filter**(lambda s: 'b' in s).**count**()
8. print "Lines with a: %i, lines with b: %i" % (numAs, numBs)

Basically, this program just counts the number of lines in ‘a’ and ‘b’ in a text file. However, we need to replace $YOUR\_SPARK\_HOME with the [**Spark’s installation**](https://data-flair.training/blogs/apache-spark-installation-on-ubuntu/) location. Moreover, we use a SparkContext to create RDDs, with the Scala and Java examples.

Now, using the bin/pyspark script, we can run this application:

1. $ cd $SPARK\_HOME
2. $ ./bin/pyspark SimpleApp1.py
3. ...
4. Lines with a: 46, Lines with b: 23

In the SparkContext constructor, we code deploy dependencies by listing them in the pyFiles option:

1. from pyspark import SparkContext
2. sc = **SparkContext**("local", "App Name", pyFiles=['MyFile.py', 'lib.zip', 'app.egg'])

Moreover, all the files which are enlisted here will be further added to the PYTHONPATH and after that, it will ship to remote worker machines. Further, we can add Code dependencies to an existing SparkContext with the help of its addPyFile() method.

Also, by passing a SparkConf object to SparkContext, we can set configuration properties:

1. from pyspark import SparkConf, SparkContext
2. conf = (**SparkConf**()
3. .**setMaster**("local")
4. .**setAppName**("My app")
5. .**set**("spark.executor.memory", "1g"))
6. sc = **SparkContext**(conf = conf)

## Comparison: Python vs Scala

* **Performance**

**Python-** In terms of performance, it is slower than Scala.

**Scala-** Scala is 10 times faster than Python.

* **Type Safety**

**Python-** It is a dynamically typed language.

**Scala-** It is Statically, typed language.

* **Ease of Use**

**Python-** Comparatively, it is less verbose and also easy in use.

**Scala-** It is highly verbose language.

* **Advanced Features**

**Python-** For [**machine learning**](https://data-flair.training/blogs/machine-learning-tutorial/) and natural language processing, Scala does not have sufficient data science tools and libraries like Python.

**Scala-** However, it has several existential types, macros, and implicit but still it lacks in visualizations and local data transformations.

# *PySpark Pros and Cons | Characteristics of PySpark?*

## 2.Advantages of PySpark

### i. Simple to write

We can say it is very simple to write parallelized code, for simple problems.

### ii. Framework handles errors

While it comes to Synchronization points as well as errors, framework easily handles them.

### iii. Algorithms

Many of the useful algorithms are already implemented in [**Spark**](https://data-flair.training/blogs/spark-tutorial/).

### iv. Libraries

In comparison to Scala, Python is far better in the available libraries. Since the huge number of libraries are available so most of the data science related parts from **[R](https://data-flair.training/blogs/r-programming-tutorial/)** are ported to Python. Well, this does not happen in the case with Scala.

### v. Good Local Tools

For Scala, there are no good visualization tools but there are some good local tools available in Python.

### vi. Learning Curve

As compared to Scala, the learning curve is less in Python.

### vii. Ease of use

Again as compared to Scala, Python is easy to use.

## 3.Disadvantages of PySpark

### i. Difficult to express

While it comes to express a problem in[**MapReduce**](https://data-flair.training/blogs/hadoop-mapreduce-tutorial/) fashion, sometimes it’s difficult.

### ii. Less Efficient

Pythons are less efficient as compared to other programming models. For example as MPI when we need a lot of communication.

### iii. slow

Basically, Python is slow as compared to Scala for Spark Jobs, Performance wise. Approximately, 10x slower. That means if we want to do heavy processing then Python will be slower than Scala.

### iv. Immature

In Spark 1.2, Python does support for Spark Streaming still it is not as mature as Scala as of now. So, we must go to Scala, if we need Streaming.

### v. Cannot use internal functioning of Spark

As the whole of Spark is written in Scala, so we have to work with Scala if we want to or have to change from internal functioning of Spark for our project, we cannot use Python for it. However, let’s understand it with an example, see using Scala in Spark core we can create a new RDD, but not create it using Python.

## 4. Characteristics of PySpark

### i. Nodes are abstracted

That means we cannot address an individual node.

### ii. Network is abstracted

Only implicit communication is possible here.

### iii. Based on Map-Reduce

Moreover, programmers offer a map and a reduce function.

### iv. API for Spark

PySpark is one of the API for Spark.

# *PySpark SparkFiles and Its Class Methods?*

## What is PySpark SparkFiles?

By using SparkFiles.get, we can upload our files in Apache [**Spark**](https://data-flair.training/blogs/apache-spark-installation-on-ubuntu/). However, sc refers to our default **[SparkContext](https://data-flair.training/blogs/pyspark-sparkcontext/)** here. Moreover, we can also get the path on a worker using the command “SparkFiles.get”. Hence, in order to resolve the paths to files added through SparkContext.addFile(), we can use SparkFiles.  
  
There are following types of class methods in SparkFiles, such as  −

* get(filename)
* getrootdirectory()

Although make sure that SparkFiles only contains class methods; users should not create SparkFiles instances.

## Class Methods of PySpark SparkFiles

So, let’s learn the two PySpark SparkFiles Class Methods in detail:

### i. get(filename)

Basically, the classmethod “get(filename)” specifies the path of the file which is added through SparkContext.addFile().

1. import os
2. -class **SparkFiles**(object):
3. """
4. Resolves paths to files added through
5. L{SparkContext.**addFile**()<pyspark.context.SparkContext.addFile>}.
6. SparkFiles contains only classmethods; users should not create SparkFiles
7. instances.
8. """
9. \_root\_directory = None
10. \_is\_running\_on\_worker = False
11. \_sc = None
12. - def **\_\_init\_\_**(self):
13. raise **NotImplementedError**("Do not construct SparkFiles objects")
14. @classmethod
15. - def **get**(cls, filename):
16. """
17. Get the absolute path of a file added through C{SparkContext.**addFile**()}.
18. """
19. path = os.path.**join**(SparkFiles.**getRootDirectory**(), filename)
20. return os.path.**abspath**(path)
21. @classmethod
22. + def **getRootDirectory**(cls):
23. ...

### ii. getrootdirectory()

Whereas, this file, specifies the path to the root directory. Basically, it contains the whole file which is added through the SparkContext.addFile().

1. import os
2. -class **SparkFiles**(object):
3. """
4. Resolves paths to files added through
5. L{SparkContext.**addFile**()<pyspark.context.SparkContext.addFile>}.
6. SparkFiles contains only classmethods; users should not create SparkFiles
7. instances.
8. """
9. \_root\_directory = None
10. \_is\_running\_on\_worker = False
11. \_sc = None
12. - def **\_\_init\_\_**(self):
13. raise **NotImplementedError**("Do not construct SparkFiles objects")
14. @classmethod
15. + def **get**(cls, filename):
16. ...
17. @classmethod
18. - def **getRootDirectory**(cls):
19. """
20. Than Get the root directory which contains files added through
21. C{SparkContext.**addFile**()}.
22. """
23. if cls.\_is\_running\_on\_worker:
24. return cls.\_root\_directory
25. else:
26. # This will have to change if we support multiple SparkContexts:
27. return cls.\_sc.\_jvm.spark.SparkFiles.**getRootDirectory**()

# *PySpark RDD With Operations and Commands?*

## What is Spark RDD?

An Acronym RDD refers to Resilient Distributed Dataset. Basically, RDD is the key abstraction of Apache [**Spark**](https://data-flair.training/blogs/spark-tutorial/). In order to do parallel processing on a cluster, these are the elements that run and operate on multiple nodes. Moreover, it is immutable in nature, that says as soon as we create an RDD we cannot change it.  
In addition, RDDs have the best feature that is “fault tolerance”. It means if any failure occurs they recover automatically.

### a. Ways to create Spark RDD

So,  to create Spark RDDs, there are 3 ways:

i. Parallelized collections  
ii. External datasets  
iii. Existing RDDs

### b. Spark RDDs operations

Moreover, to achieve a certain task, we can apply multiple operations on these RDDs.

**i. Transformation Operations:**

Transformation Operations creates a new Spark RDD from the existing one. In addition, this passes the dataset to the function and then returns new dataset as a result.

**ii. Action Operations:**

And, this operation returns final result to driver program or also writes it to the external data store.

## 3. How to Create PySpark RDD?

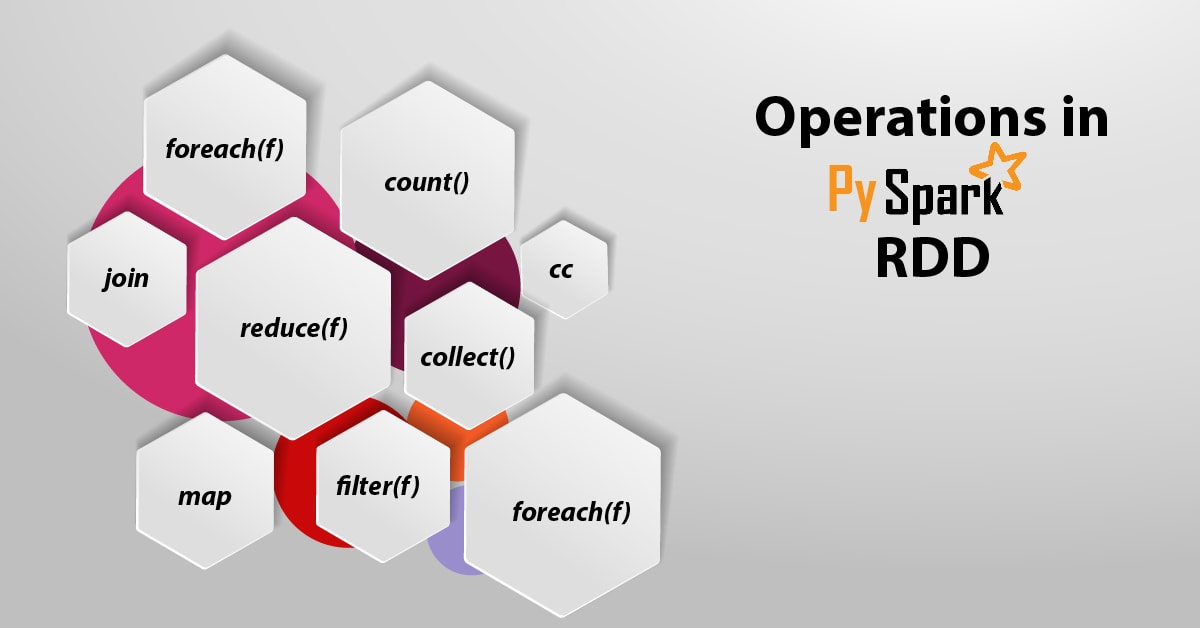
At very first, we need to create a PySpark RDD to apply any operation in PySpark. For that, here is a code block which has the full detail of a PySpark RDD Class −

1. class pyspark.**RDD** (
2. jrdd,
3. ctx,
4. jrdd\_deserializer = **AutoBatchedSerializer**(**PickleSerializer**())
5. )

Further, let’s see the way to run a few basic operations using PySpark. So, here is the following code in a Python file creates RDD words, basically, that stores a set of words which is mentioned here.

1. words = sc.**parallelize** (
2. ["scala",
3. "java",
4. "hadoop",
5. "spark",
6. "akka",
7. "spark vs hadoop",
8. "pyspark",
9. "pyspark and spark"]
10. )

## 4. Operations in PySpark RDD

[](https://d2h0cx97tjks2p.cloudfront.net/blogs/wp-content/uploads/sites/2/2018/06/Operations-in-PySpark-RDD-01.jpg)

### i. count()

With this operation, the number of elements in the RDD is returned.

1. ----------------------------------------count.py---------------------------------------
2. from pyspark import SparkContext
3. sc = **SparkContext**("local", "count app")
4. words = sc.**parallelize** (
5. ["scala",
6. "java",
7. "hadoop",
8. "spark",
9. "akka",
10. "spark vs hadoop",
11. "pyspark",
12. "pyspark and spark"]
13. )
14. counts = words.**count**()
15. print "Number of elements in RDD -> %i" % (counts)
16. ----------------------------------------count.py---------------------------------------

#### ****Command****

1. $SPARK\_HOME/bin/spark-submit count.py

**Outcome**  
**Number of elements in RDD → 8**

### ii. collect()

Basically, this operation returns all the elements in the RDD.

1. ---------------------------------------collect.py---------------------------------------
2. from pyspark import SparkContext
3. sc = **SparkContext**("local", "Collect app")
4. words = sc.**parallelize** (
5. ["scala",
6. "java",
7. "hadoop",
8. "spark",
9. "akka",
10. "spark vs hadoop",
11. "pyspark",
12. "pyspark and spark"]
13. )
14. coll = words.**collect**()
15. print "Elements in RDD -> %s" % (coll)
16. ----------------------------------------collect.py---------------------------------------

**Command**

1. $SPARK\_HOME/bin/spark-submit collect.py

## **Output** Elements in RDD -> [ ‘scala’,https://data-flair.training/blogs/python-tutorial-for-beginners/  ‘java’,  ‘hadoop’,  ‘spark’,  ‘akka’,  ‘spark vs hadoop’,  ‘pyspark’,  ‘pyspark and spark’ ]

### iii. foreach(f)

foreach(f) operations returns only those elements which meet the condition of the function inside foreach. Here, to prints all the elements in the RDD, we will call a print function in foreach.

1. ----------------------------------------foreach.py---------------------------------------
2. from pyspark import SparkContext
3. sc = **SparkContext**("local", "ForEach app")
4. words = sc.**parallelize** (
5. ["scala",
6. "java",
7. "hadoop",
8. "spark",
9. "akka",
10. "spark vs hadoop",
11. "pyspark",
12. "pyspark and spark"]
13. )
14. def **f**(x): **print**(x)
15. fore = words.**foreach**(f)
16. ----------------------------------------foreach.py---------------------------------------

**Command**

1. $SPARK\_HOME/bin/spark-submit foreach.py

Output  
scala  
java  
hadoop  
spark  
akka  
spark vs hadoop  
pyspark  
pyspark and spark

### iv. cc

After applying this operation, we will get a new RDD which contains the elements, those satisfy the function inside the filter. Now, here we filter out the strings containing ”spark”, in the following example.

1. ----------------------------------------filter.py---------------------------------------
2. from pyspark import SparkContext
3. sc = **SparkContext**("local", "Filter app")
4. words = sc.**parallelize** (
5. ["scala",
6. "java",
7. "hadoop",
8. "spark",
9. "akka",
10. "spark vs hadoop",
11. "pyspark",
12. "pyspark and spark"]
13. )
14. words\_filter = words.**filter**(lambda x: 'spark' in x)
15. filtered = words\_filter.**collect**()
16. print "Fitered RDD -> %s" % (filtered)
17. ----------------------------------------filter.py----------------------------------------

**Command**

1. $SPARK\_HOME/bin/spark-submit filter.py

#### **Output**

Fitered RDD -> [  
‘spark’,   
‘spark vs hadoop’,   
‘pyspark’,   
‘pyspark and spark’  
]

### v. map(f, preservesPartitioning = False)

By applying a function to each element in the RDD, a new RDD is returned. Now, here, we form a key-value pair and map every string with a value of 1 in the following example.

1. ----------------------------------------map.py---------------------------------------
2. from pyspark import SparkContext
3. sc = **SparkContext**("local", "Map app")
4. words = sc.**parallelize** (
5. ["scala",
6. "java",
7. "hadoop",
8. "spark",
9. "akka",
10. "spark vs hadoop",
11. "pyspark",
12. "pyspark and spark"]
13. )
14. words\_map = words.**map**(lambda x: (x, 1))
15. mapping = words\_map.**collect**()
16. print "Key value pair -> %s" % (mapping)
17. ----------------------------------------map.py---------------------------------------

Command

1. $SPARK\_HOME/bin/spark-submit map.py

Output  
Key value pair -> [  
(‘scala’, 1),   
(‘java’, 1),   
(‘hadoop’, 1),   
(‘spark’, 1),   
(‘akka’, 1),   
(‘spark vs hadoop’, 1),   
(‘pyspark’, 1),   
(‘pyspark and spark’, 1)  
]

### vi. reduce(f)

Here, the element in the RDD is returned, after performing the specified commutative and associative binary operation. So, we are importing add package from the operator and also to carry out a simple addition operation we are applying it on ‘num’, in the following example.

1. ----------------------------------------reduce.py---------------------------------------
2. from pyspark import SparkContext
3. from operator import add
4. sc = **SparkContext**("local", "Reduce app")
5. nums = sc.**parallelize**([1, 2, 3, 4, 5])
6. adding = nums.**reduce**(add)
7. print "Adding all the elements -> %i" % (adding)
8. ----------------------------------------reduce.py---------------------------------------

Command

1. $SPARK\_HOME/bin/spark-submit reduce.py

Output  
Adding all the elements -> 15

### vii. join(other, numPartitions = None)

This operation returns RDD with a pair of elements with the matching keys as well as all the values for that particular key. So, there is two pair of elements in two different RDDs, in the following example. So, we get an RDD with elements having matching keys and their values, after joining these two RDDs.

1. ----------------------------------------join.py---------------------------------------
2. from pyspark import SparkContext
3. sc = **SparkContext**("local", "Join app")
4. x = sc.**parallelize**([("spark", 1), ("hadoop", 4)])
5. y = sc.**parallelize**([("spark", 2), ("hadoop", 5)])
6. joined = x.**join**(y)
7. final = joined.**collect**()
8. print "Join RDD -> %s" % (final)
9. ----------------------------------------join.py---------------------------------------

Command

1. $SPARK\_HOME/bin/spark-submit join.py

Output  
Join RDD -> [  
(‘spark’, (1, 2)),   
(‘hadoop’, (4, 5))  
]

### viii. cache()

Moreover, this command, with the default storage level (MEMORY\_ONLY), Persist this RDD. Also, we can check if the RDD is cached or not with cache() command

1. ----------------------------------------cache.py---------------------------------------
2. from pyspark import SparkContext
3. sc = **SparkContext**("local", "Cache app")
4. words = sc.**parallelize** (
5. ["scala",
6. "java",
7. "hadoop",
8. "spark",
9. "akka",
10. "spark vs hadoop",
11. "pyspark",
12. "pyspark and spark"]
13. )
14. words.**cache**()
15. caching = words.**persist**().is\_cached
16. print "Words got chached > %s" % (caching)
17. ----------------------------------------cache.py---------------------------------------

Command

1. $SPARK\_HOME/bin/spark-submit cache.py

Output  
Words got cached -> True

## *What is PySpark SparkConf?*

We need to set a few configurations and parameters, to run a Spark application on the local/cluster, this is what SparkConf helps with. Basically,  to run a Spark application, it offers configurations.

* **Code**

For PySpark, here is the code block which has the details of a SparkConf class:

1. class pyspark.**SparkConf** (
2. loadDefaults = True,
3. \_jvm = None,
4. \_jconf = None
5. )

Basically, with SparkConf() we will create a SparkConf object first. So, that will load the values from spark. Even [**Java**](https://data-flair.training/blogs/java-tutorial/) system properties. Hence, by using the SparkConf object, now we can set different parameters and their parameters will take priority over the system properties.  
However, there are better methods, which support chaining, in a SparkConf class. Let’s say, we can write conf.setAppName(“PySpark App”).setMaster(“local”). Though, it cannot be modified by any user once we pass a SparkConf object to Apache Spark.

## 3. Attributes of PySpark SparkConf

Thus here are the most commonly used attributes of SparkConf:

### Attributes of PySpark SparkConfi. set(key, value)

It helps to set a configuration property.

### ii. setMaster(value)

In order to set the master URL, we use it.

### iii. setAppName(value)

We use it to set an application name.

### iv. get(key, defaultValue=None)

It helps to get a configuration value of a key.

### v. setSparkHome(value)

In order to set [**Spark installation**](https://data-flair.training/blogs/apache-spark-installation-on-ubuntu/) path on worker nodes, we use it.  
In the following code, we can use to create SparkConf and **[SparkContext](https://data-flair.training/blogs/pyspark-sparkcontext/)**objects as part of our applications. Also, using sbt console on base directory of our application we can validate:

1. from pyspark import SparkConf,SparkContext
2. conf = **SparkConf**().**setAppName**("Spark Demo").**setMaster**("local")
3. sc = **SparkContext**(conf=conf)

**4. Running Spark Applications Using SparkConf**

In addition, here are some different contexts in which we can run spark applications:

* **local – conf**

SparkConf.setAppName(“Spark Demo”).setMaster(“local”)

* **yarn-client – conf**

SparkConf.setAppName(“Spark Demo”).setMaster(“yarn-client”)

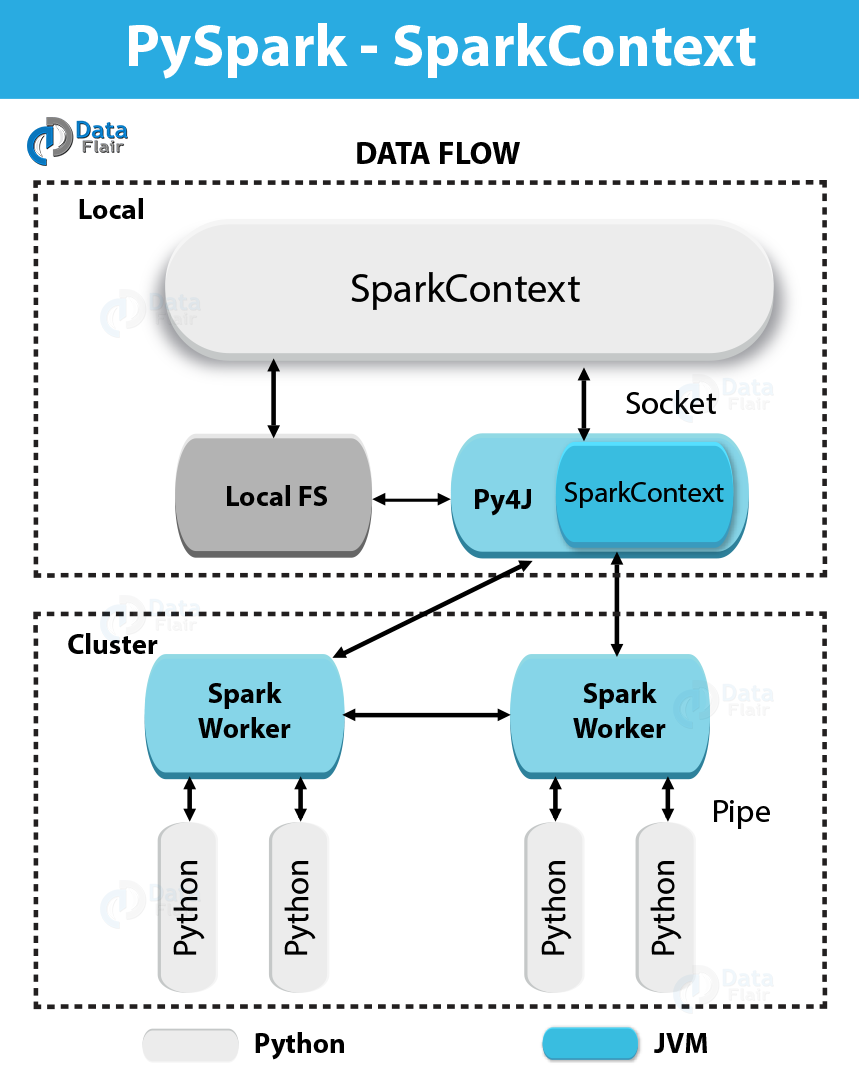
* **mesos URL**
* **spark URL – conf**

SparkConf.setAppName(“Spark Demo”).setMaster(“spark master URL”)

* **Code snippet to get all the properties**

for i in sc.getConf.getAll: print(i)

***What is SparkContext in PySpark?***

In simple words, an entry point to any [**Spark**](https://data-flair.training/blogs/spark-tutorial/) functionality is what we call SparkContext. At the time we run any Spark application, a driver program starts, which has the main function and from this time your SparkContext gets initiated. Afterward, on worker nodes, driver program runs the operations inside the executors.  
In addition, to launch a [**JVM**](https://data-flair.training/blogs/java-virtual-machine-jvm/), SparkContext uses Py4J and then creates a JavaSparkContext. However, PySpark has SparkContext available as ‘sc’, by default, thus the creation of a new SparkContext won’t work.

## 3. Parameters in PySpark SparkContext

### a. Master

This is the URL of the cluster it connects to.

### b. appName

Basically, “appName” parameter refers to the name of your job.

### c. SparkHome

Generally, sparkHome is a [**Spark installation**](https://data-flair.training/blogs/apache-spark-installation-on-ubuntu/) directory.

### d. pyFiles

Files like .zip or .py files are to send to the cluster and to add to the PYTHONPATH.

### e. Environment

Worker nodes environment variables.

### f. BatchSize

Basically, as a single [**Java object**](https://data-flair.training/blogs/class-and-object-in-java/), the number of [**Python objects**](https://data-flair.training/blogs/python-classes/) represented. However, to disable batching, Set 1, and to automatically choose the batch size based on object sizes set 0, Also, to use an unlimited batch size, set -1.

### g. Serializer

So, this parameter tell about Serializer, an RDD serializer.

### h. Conf

Moreover, to set all the Spark properties, an object of L{SparkConf} is there.

### i. Gateway

Basically, use an existing gateway as well as JVM, else initialize a new JVM.

### j. JSC

However, JSC is the JavaSparkContext instance.

### k. profiler\_cls

Basically, in order to do profiling, a class of custom Profiler is used. Although, make sure the pyspark.profiler.BasicProfiler is the default one.

So, master and appname are mostly used, among the above parameters. However, any PySpark program’s first two lines look as shown below −

1. from pyspark import SparkContext
2. sc = **SparkContext**("local", "First App1")

## 4. SparkContext Example – PySpark Shell

Since we have learned much about PySpark SparkContext, now let’s understand it with an example. Here we will count the number of the lines with character ‘x’ or ‘y’ in the README.md file. So, let’s assume that there are 5 lines in a file. Hence, 3 lines have the character ‘x’, then the output will be → Line with x: 3. However, for character ‘y’, same will be done .

However, make sure in the following PySpark SparkContext example we are not creating any SparkContext object it is because Spark automatically creates the SparkContext object named sc, by default, at the time PySpark shell starts. So, If you try to create another SparkContext object, following error will occur – “ValueError: That says, it is not possible to run multiple SparkContexts at once”.

1. <<< logFile = "file:///home/hadoop/spark-2.1.0-bin-hadoop2.7/README.md"
2. <<< logData = sc.**textFile**(logFile).**cache**()
3. <<< numXs = logData.**filter**(lambda s: 'x' in s).**count**()
4. <<< numYs = logData.**filter**(lambda s: 'y' in s).**count**()
5. <<< print "Lines with x: %i, lines with y: %i" % (numXs, numYs)
6. Lines with x: 62, lines with y: 30

## 5. SparkContext Example – Python Program

1. ----------------------------------------firstapp1.py---------------------------------------
2. from pyspark import SparkContext
3. logFile = "file:///home/hadoop/spark-2.1.0-bin-hadoop2.7/README.md"
4. sc = **SparkContext**("local", "first app")
5. logData = sc.**textFile**(logFile).**cache**()
6. numXs = logData.**filter**(lambda s: 'x' in s).**count**()
7. numYs = logData.**filter**(lambda s: 'y' in s).**count**()
8. print "Lines with x: %i, lines with y: %i" % (numXs, numYs)
9. ----------------------------------------firstapp1.py---------------------------------------

Then to run this Python file, we will execute the following command in the terminal.  
Hence, it will give the same **output** as above:

1. $SPARK\_HOME/bin/spark-submit firstapp1.py

**Output: Lines with x: 62, lines with y: 30**

<https://data-flair.training/blogs/pyspark-sparkconf/>