Introduction to Apache Spark

What is Apache Spark

- Apache Spark is a fast and general-purpose cluster (big data) computing system and completely interoperable with Hadoop (read data from HDFS, Hive, …).
- It was designed to be a generalization of Map/Reduce. Repalcing the Map/Reduce side in the traditional Hadoop Map/Reduace system can be up to 100x faster in memory or 10x faster on disk and needs less coding.
- It provides high-level APIs in **Java**, **Scala**, **Python**, etc.
- It also supports many high-level tools (such as Spark SQL for SQL and structured data processing and MLlib for machine learning) and batch, interactive analytics.

The First Abstraction of Spark

- The first abstraction of Spark is the **resilient distributed dataset (RDD)**, an immutable collection of elements partitioned across cluster nodes that can be operated on in parallel.
- At a high level, a Spark application consists of a **driver program** that runs the user's main function and executes various parallel operations on a cluster.
- RDDs are created by starting with a file in the HDFS (or any other Hadoop-supported file systems), and transforming it.
- An RDD may persist in memory, allowing it to be reused efficiently across parallel operations.
- RDDs can automatically recover from node failures.

The Second Abstraction of Spark

- The second abstraction in Spark is **shared variables** that can be used in parallel operations.
- By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task.
- Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. Spark supports two types of shared variables:
  - **broadcast variables**, variables to cache values in memory on all nodes.
  - **accumulators**, variables that are only "added" to (e.g., counters and sums).

Two Types of RDDs

- Parallelized collections, which take an existing (Scala) collection and run functions on it in parallel.
- Hadoop datasets, which run functions on each record of a file in HDFS or any other storage systems supported by Hadoop, such as the local file system, Amazon S3, Hypertable, Hbase, etc.
- Both types of RDDs can be operated on through the same methods.

RDD Operations

RDDs support two types of operations:

- **Transformations**, which create a new dataset from an existing one.
- **Actions**, which return a value to the driver program after running a computation on the dataset.

RDD Transformations

- All transformations are **lazy**, meaning that they do not compute their results right away, instead they just remember the transformations applied to some dataset. The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently.
- By default, each transformed RDD is recomputed each time you run an action on it. However, you may also persist an RDD in memory, on disk or replicated across the cluster.

A List of RDD Transformation Functions

- **map**
- **filter**(func)
- **flatMap**(func)
- **mapPartitions**(func)
- **mapPartitionsWithIndex**(func)
- **sample**(withReplacement, fraction, seed)
- **union**(otherDataset)

Continued …

A List of RDD Transformation Functions

- **groupByKey**([numTasks])
- **reduceByKey**(func, [numTasks])
- **sortByKey**([ascending], [numTasks])
- **join**(otherDataset, [numTasks])
- **cogroup**(otherDataset, [numTasks])

A List of RDD Action Functions

- **reduce**(func)
- **collect**()
- **count**()
- **first**()
- **take**(n)
- **takeSample**(withReplacement, num, seed)
- **countByKey**()
- **foreach**(func)
- **saveAsTextFile**(path)
- **saveAsSequenceFile**(path)

RDD Persistence

RDD persistence: persisting (or caching) a dataset in memory across operations to reuse it in other actions. This allows future actions to be much faster. It is a key tool for building iterative algorithms with Spark.

Use the **persist()** or **cache()** methods to mark an RDD to be persisted. If any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it (fault-tolerant).

An RDD can be stored at different levels: memory, disk, or nodes (e.g., MEMORY_ONLY, DISK_ONLY).

PySpark

PySpark is the Spark Python API, which exposes the Spark programming model to Python. It contains the following subpackages:

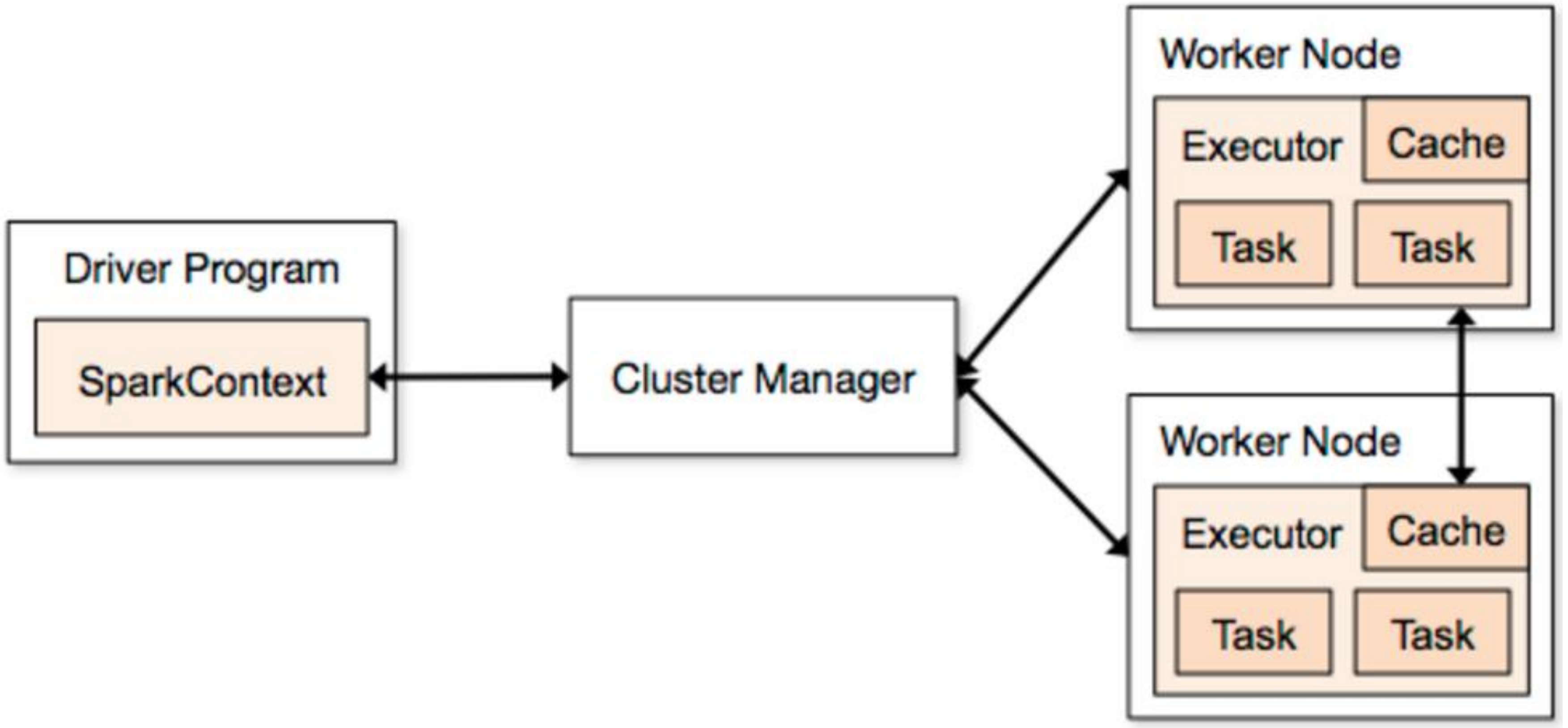
- pyspark.sql module
- pyspark.streaming module
- pyspark.ml package
- pyspark.mllib package

PySpark Public Classes

SparkContext: Main entry point for Spark functionality.
RDD: A resilient distributed dataset (RDD), the basic abstraction in Spark.
Broadcast: A broadcast variable that gets reused across tasks.
Accumulator: An "add-only" shared variable that tasks can only add values to.
SparkConf: For configuring Spark.
SparkFiles: Access files shipped with jobs.
StorageLevel: Cache persistence levels.

SparkContext Architecture

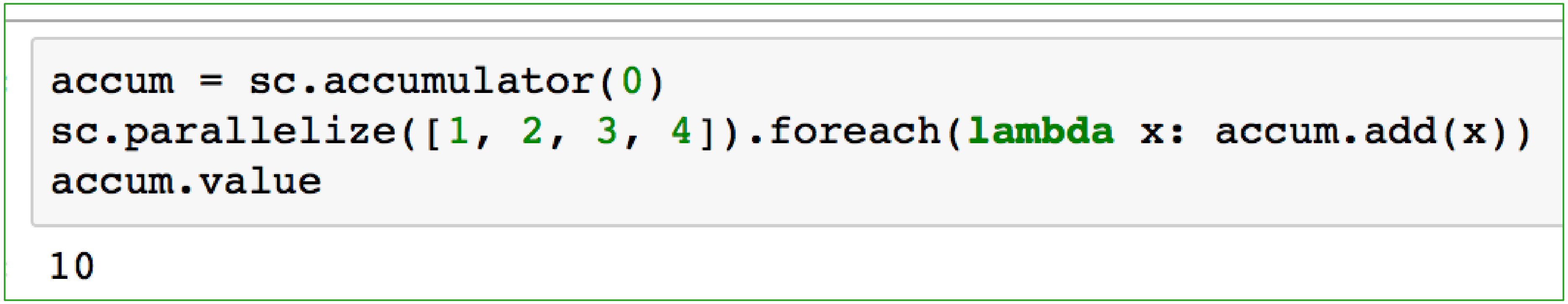
- The first thing a Spark program must do is to create a SparkContext object to tell Spark how to access a cluster.
- A SparkContext object connects to the cluster managers, which manage the actual executors that run specific computations.



Class *PySpark.SparkContext*

A SparkContext represents the connection to a Spark cluster and can be used to create RDD and broadcast variables.

- **accumulator**(value, *accum_param=None*): Creates an Accumulator with the given initial value, using a given AccumulatorParam helper object to define how to add values of the data type if provided.



```
accum = sc.accumulator(0)
sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
accum.value
```

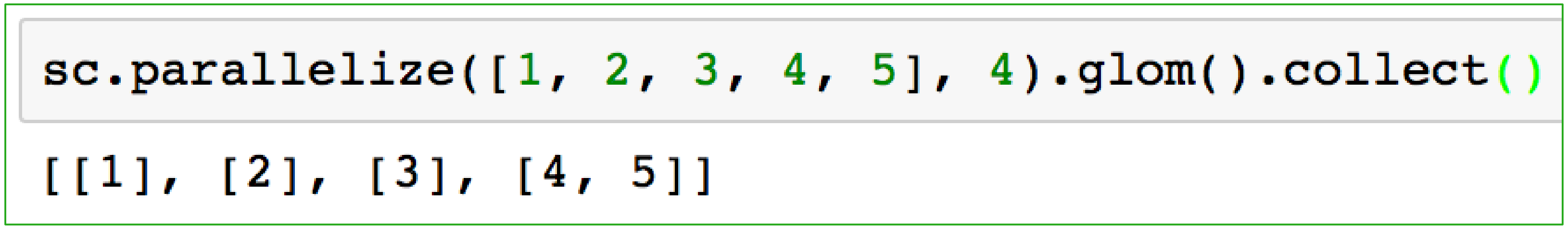
```
10
```

Continued …

Class *PySpark.SparkContext*

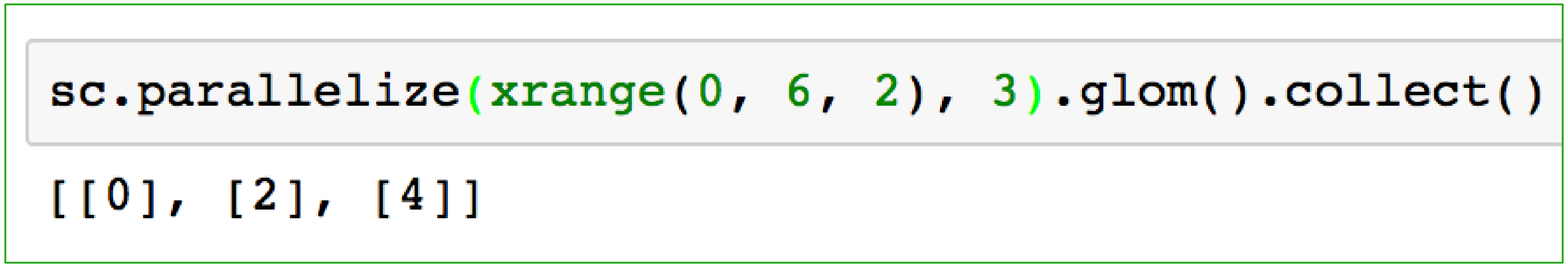
A SparkContext represents the connection to a Spark cluster and can be used to create RDD and broadcast variables.

- **parallelize**(c, *numSlices=None*): Distributes a local Python collection to form an RDD. Using **xrange** is recommended if the input represents a range for performance.



```
sc.parallelize([1, 2, 3, 4, 5], 4).glom().collect()
```

```
[[1], [2], [3], [4, 5]]
```



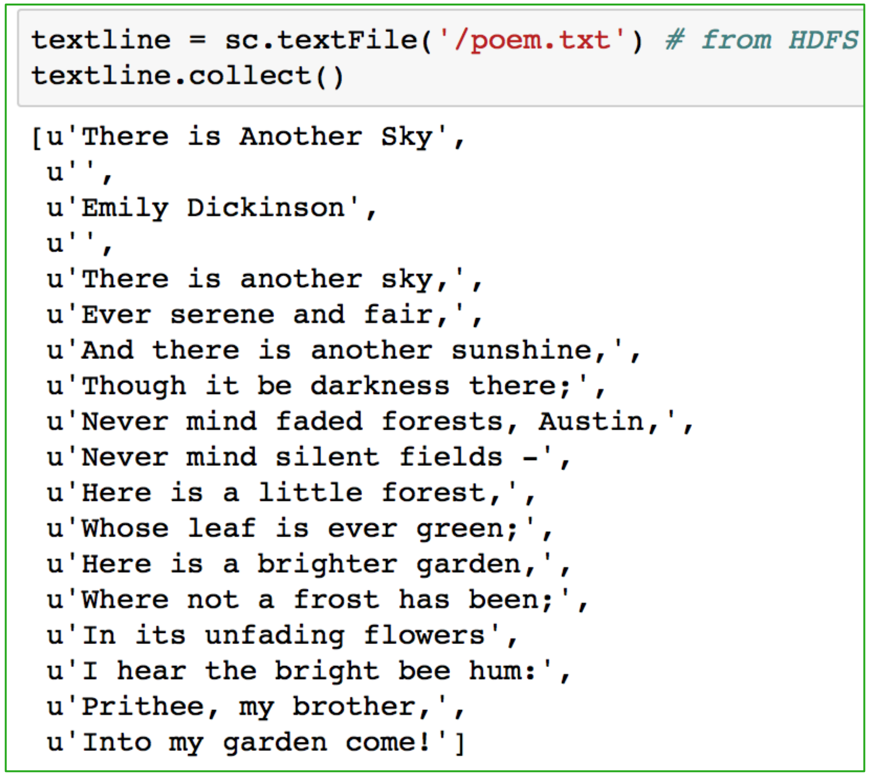
```
sc.parallelize(xrange(0, 6, 2), 3).glom().collect()
```

```
[[0], [2], [4]]
```

Class *PySpark.SparkContext*

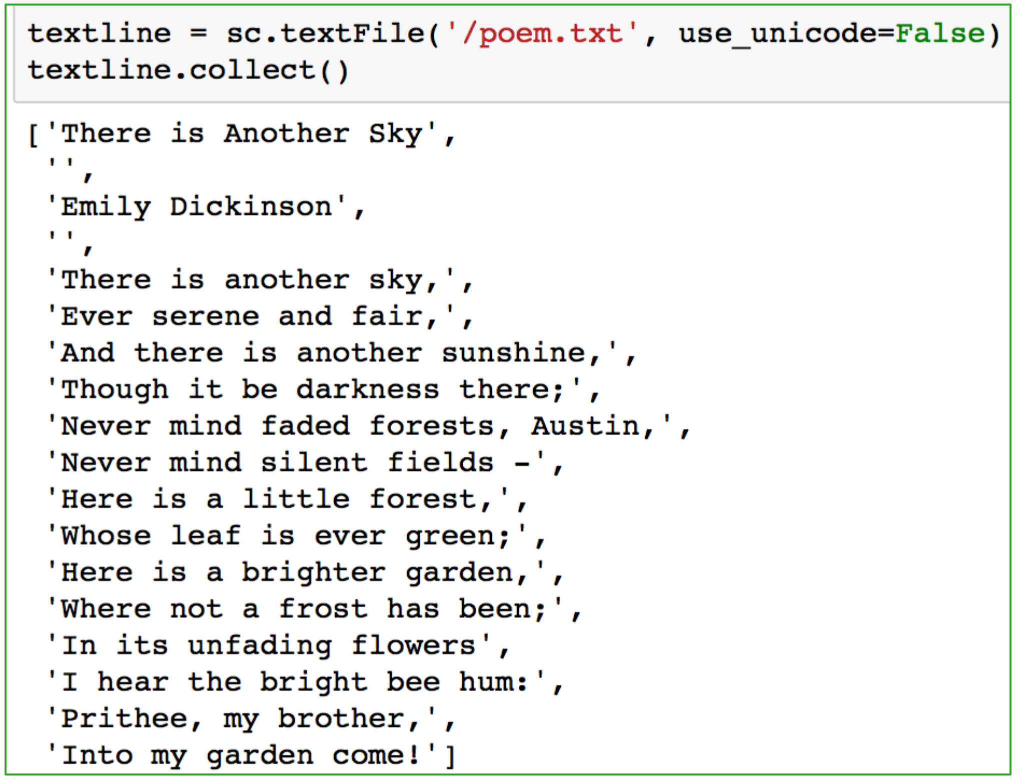
textFile(name, *minPartitions=None, use_unicode=True*):

Reads a text file from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI, and returns it as an RDD of Strings.



```
textline = sc.textFile('/poem.txt') # from HDFS
textline.collect()
```

```
[u'There is Another Sky',
 u'',
 u'Emily Dickinson',
 u'',
 u'There is another sky,',
 u'Ever serene and fair,',
 u'And there is another sunshine,',
 u'Though it be darkness there;',
 u'Never mind faded forests, Austin,',
 u'Never mind silent fields -',
 u'Here is a little forest,',
 u'Whose leaf is ever green;',
 u'Here is a brighter garden,',
 u'Where not a frost has been;',
 u'In its unfading flowers',
 u'I hear the bright bee hum:',
 u'Prithee, my brother,',
 u'Into my garden come!']
```



```
textline = sc.textFile('/poem.txt', use_unicode=False)
textline.collect()
```

```
['There is Another Sky',
 '',
 'Emily Dickinson',
 '',
 'There is another sky,',
 'Ever serene and fair,',
 'And there is another sunshine,',
 'Though it be darkness there;',
 'Never mind faded forests, Austin,',
 'Never mind silent fields -',
 'Here is a little forest,',
 'Whose leaf is ever green;',
 'Here is a brighter garden,',
 'Where not a frost has been;',
 'In its unfading flowers',
 'I hear the bright bee hum:',
 'Prithee, my brother,',
 'Into my garden come!']
```

PySpark RDD Methods and Functions

RDD methods take a function as an argument. In PySpark, RDDs support the same methods as their Scala counterparts but take Python functions and return Python collection types.

- Short functions can be passed to RDD methods using Python's **lambda** syntax.

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

- Python functions defined with the **def** keyword can be passed as well. It is useful for longer functions that can't be expressed using the lambda syntax.

```
def has_error(line):
    return "ERROR" in line
errors = logData.filter(has_error)
```

Class *PySpark.RDD*

cogroup(other, *numPartitions=None*): For each key k in self or other, returns a resulting RDD that contains a tuple with the list of values for that key in self as well as other.

```
x = sc.parallelize([("a", 10), ("b", 20)])
y = sc.parallelize([("b", 30)])
[(x, tuple(map(list, y))) for x, y in sorted(list(x.cogroup(y).collect()))]
```

```
[('a', ([10], [])), ('b', ([20], [30]))]
```

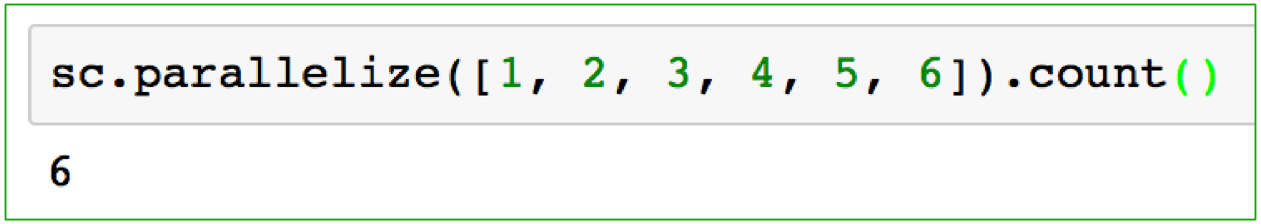
collect(): Returns a list that contains all of the elements in this RDD.

```
sc.range(5).collect()
```

```
[0, 1, 2, 3, 4]
```

Class *PySpark.RDD*

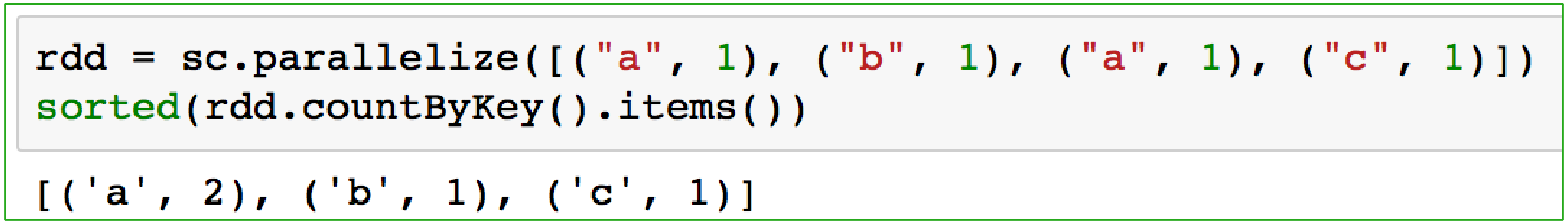
count(): Returns the number of elements in this RDD.



```
sc.parallelize([1, 2, 3, 4, 5, 6]).count()
```

```
6
```

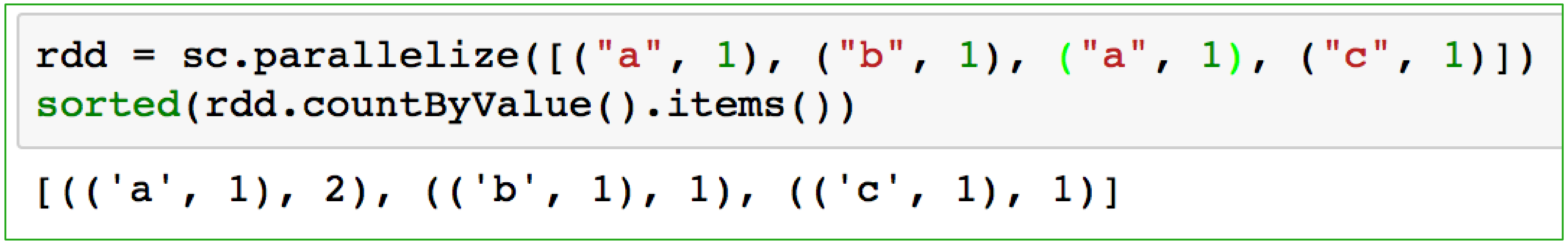
countByKey(): Counts the number of elements for each key.



```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1), ("c", 1)])
sorted(rdd.countByKey().items())
```

```
[('a', 2), ('b', 1), ('c', 1)]
```

countByValue(): Returns the count of each unique value in this RDD as (value, count) pairs.

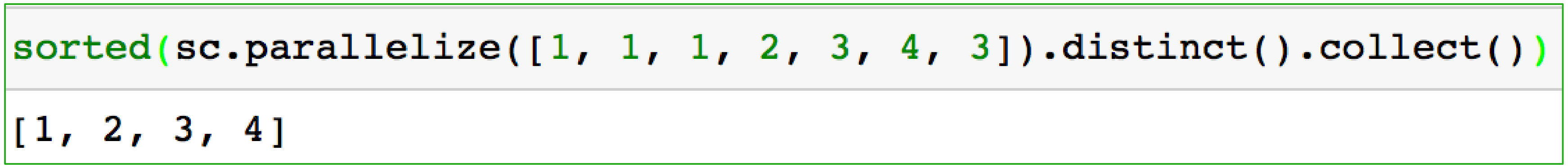


```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1), ("c", 1)])
sorted(rdd.countByValue().items())
```

```
[(('a', 1), 2), (('b', 1), 1), (('c', 1), 1)]
```

Class *PySpark.RDD*

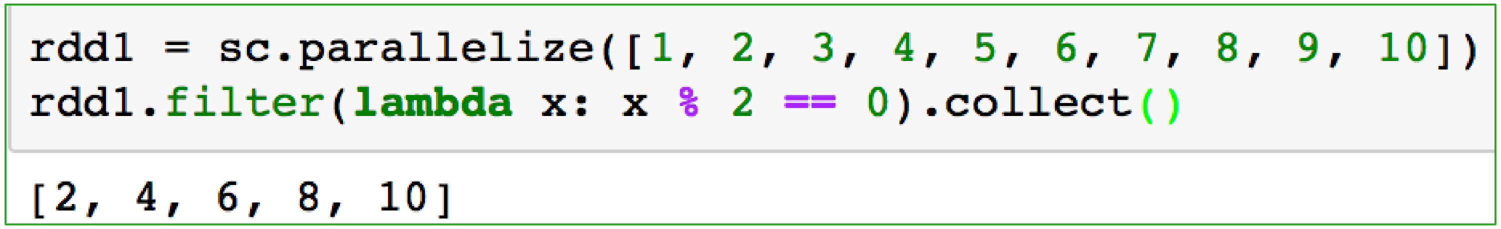
distinct(*numPartitions=None*): Returns a new RDD containing the distinct elements in this RDD.



```
sorted(sc.parallelize([1, 1, 1, 2, 3, 4, 3]).distinct().collect())
```

```
[1, 2, 3, 4]
```

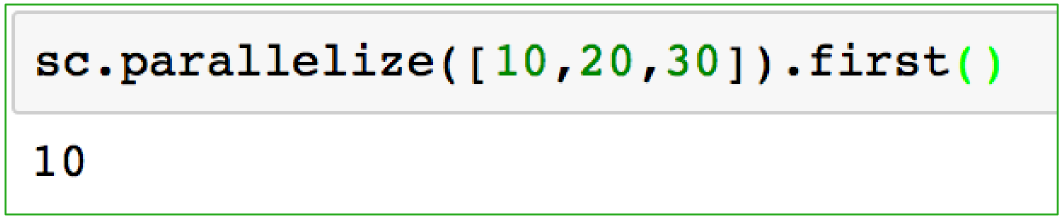
filter(f): Returns a new RDD containing only the elements that satisfy a predicate.



```
rdd1 = sc.parallelize([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
rdd1.filter(lambda x: x % 2 == 0).collect()
```

```
[2, 4, 6, 8, 10]
```

first(): Returns the first element in this RDD.

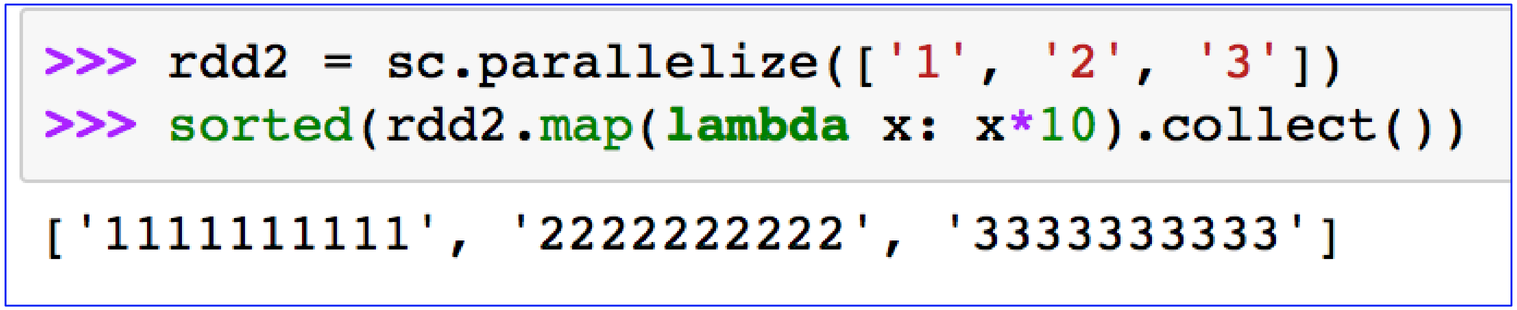


```
sc.parallelize([10,20,30]).first()
```

```
10
```

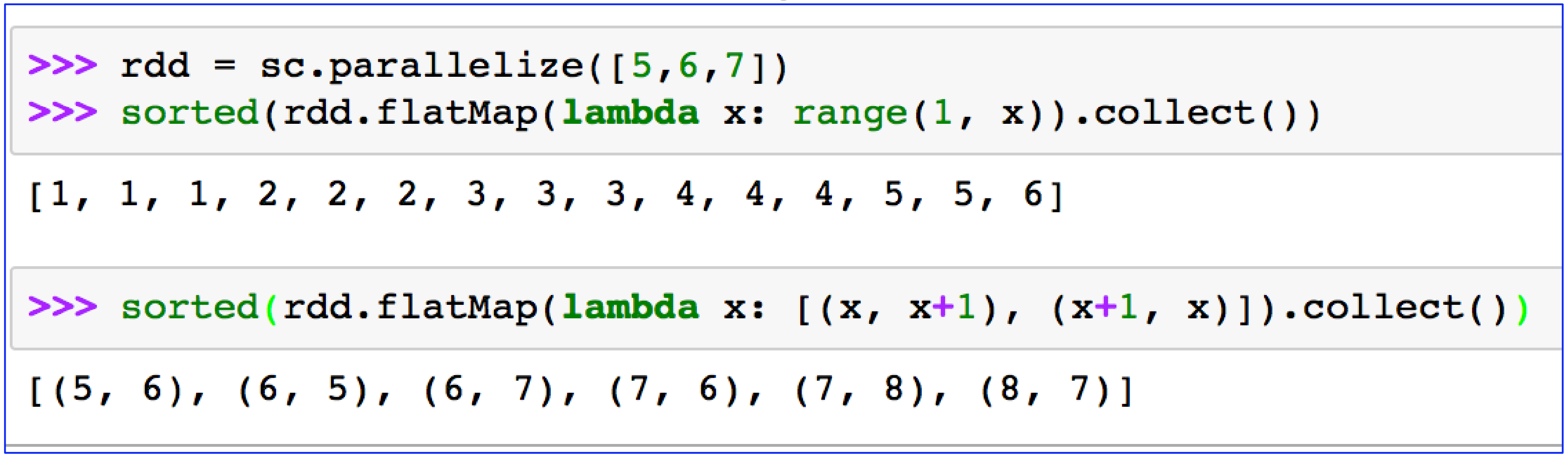
Class *PySpark.RDD*

map(f, *preservesPartitioning=False*): Returns a new RDD by applying a function to each element of this RDD.



```
>>> rdd2 = sc.parallelize(['1', '2', '3'])
>>> sorted(rdd2.map(lambda x: x*10).collect())

['1111111111', '2222222222', '3333333333']
```

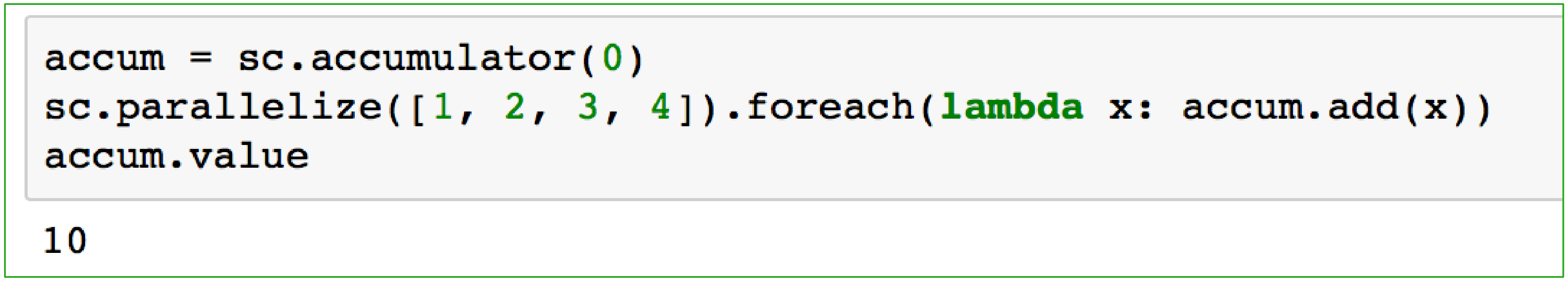
flatMap(f, *preservesPartitioning=False*): Returns a new RDD by first applying a function to all elements of this RDD, and then flattening the results.



```
>>> rdd = sc.parallelize([5,6,7])
>>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())

[1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 6]

>>> sorted(rdd.flatMap(lambda x: [(x, x+1), (x+1, x)]).collect())

[(5, 6), (6, 5), (6, 7), (7, 6), (7, 8), (8, 7)]
```

Class *PySpark.RDD*

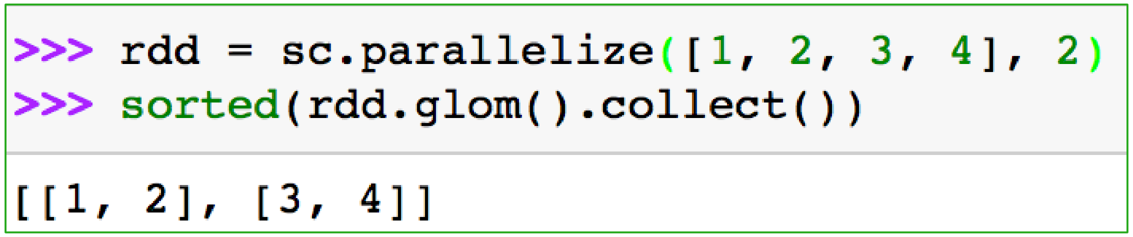
foreach(f): Applies a function to all elements of this RDD.



```
accum = sc.accumulator(0)
sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
accum.value
```

```
10
```

glom(): Returns an RDD created by coalescing all elements within each partition into a list.

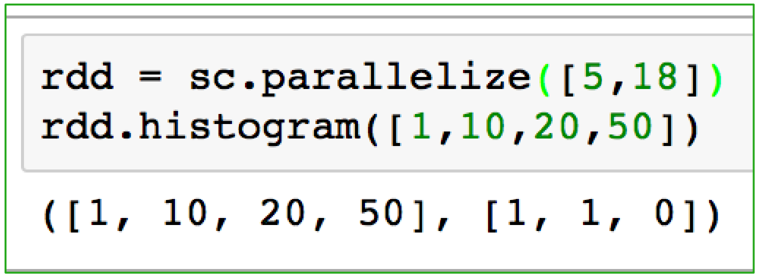


```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> sorted(rdd.glom().collect())
```

```
[[1, 2], [3, 4]]
```

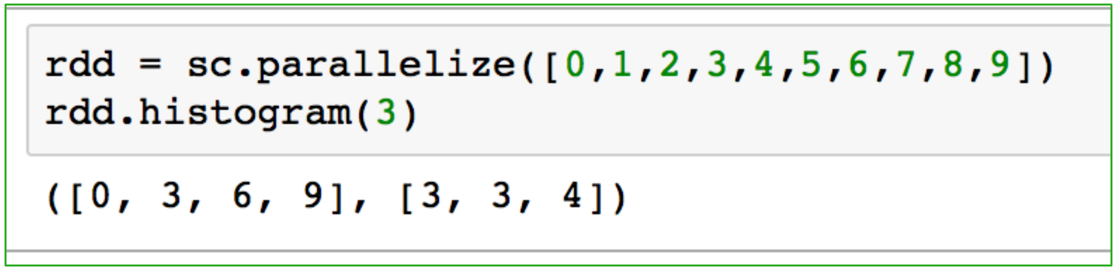
Class *PySpark.RDD*

histogram(buckets): Computes a histogram using the buckets, which are all open to the right except for the last which is closed, e.g., [1,10,20,50] means the buckets are [1,10) [10,20) [20,50].



```
rdd = sc.parallelize([5,18])
rdd.histogram([1,10,20,50])

([1, 10, 20, 50], [1, 1, 0])
```

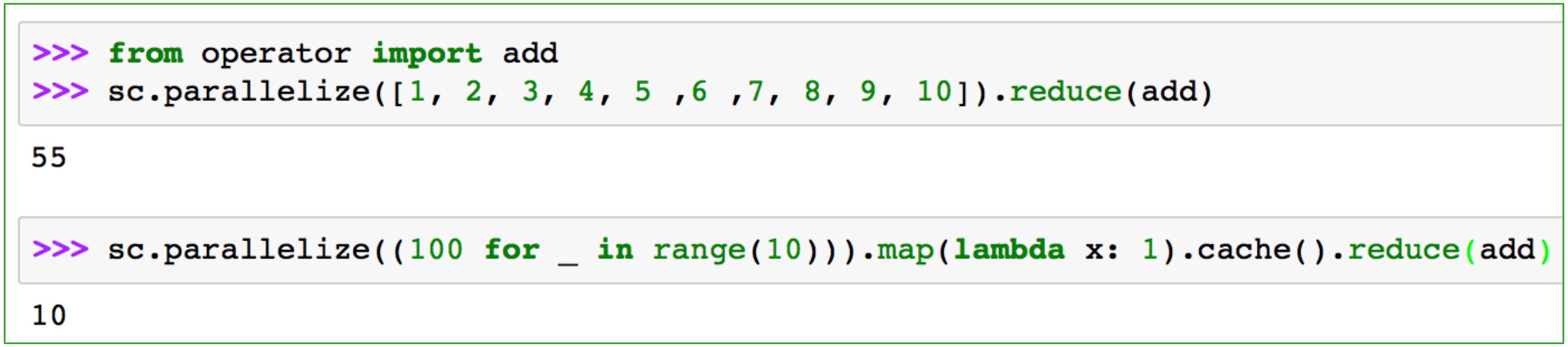
- Buckets must be sorted and not contain any duplicates, must be at least two elements.
- If "buckets" is a number, it will generates buckets which are evenly spaced between the minimum and maximum of the RDD.



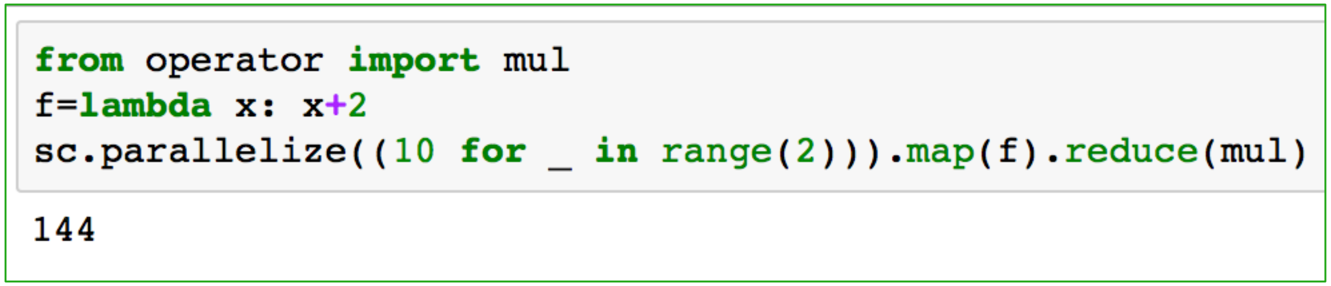
```
rdd = sc.parallelize([0,1,2,3,4,5,6,7,8,9])
rdd.histogram(3)

([0, 3, 6, 9], [3, 3, 4])
```

Class *PySpark.RDD*

reduce(f): Reduces the elements of this RDD using the specified commutative and associative binary operator.



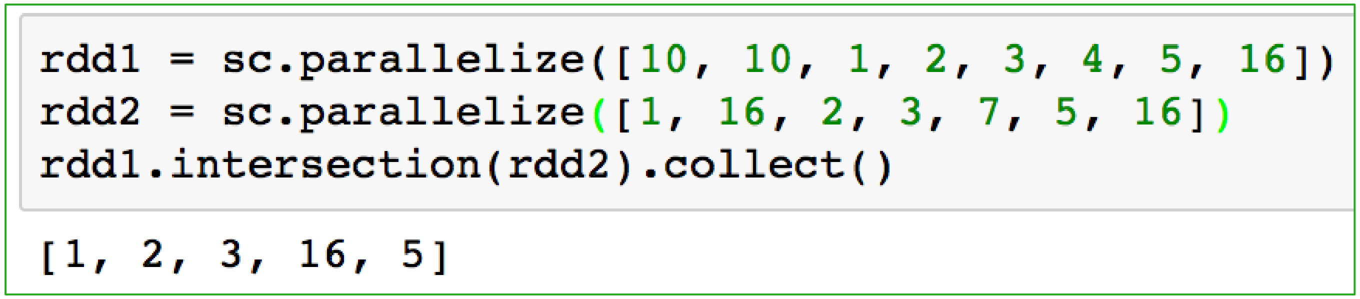
```
>>> from operator import add
>>> sc.parallelize([1, 2, 3, 4, 5 ,6 ,7, 8, 9, 10]).reduce(add)

55

>>> sc.parallelize((100 for _ in range(10))).map(lambda x: 1).cache().reduce(add)

10
```



```
from operator import mul
f=lambda x: x+2
sc.parallelize((10 for _ in range(2))).map(f).reduce(mul)

144
```

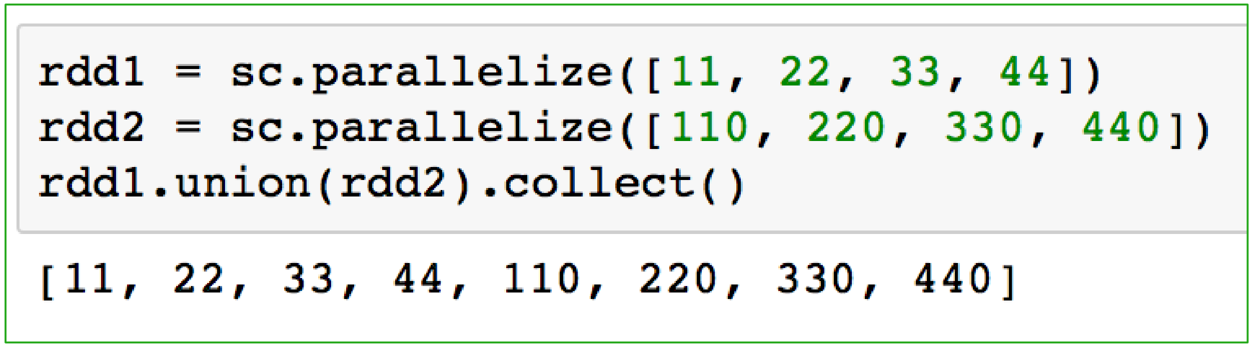
Class *PySpark.RDD*

intersection(other RDD): Returns the intersection of this RDD and another RDD. The output does not contain duplicate elements.



```
rdd1 = sc.parallelize([10, 10, 1, 2, 3, 4, 5, 16])
rdd2 = sc.parallelize([1, 16, 2, 3, 7, 5, 16])
rdd1.intersection(rdd2).collect()

[1, 2, 3, 16, 5]
```

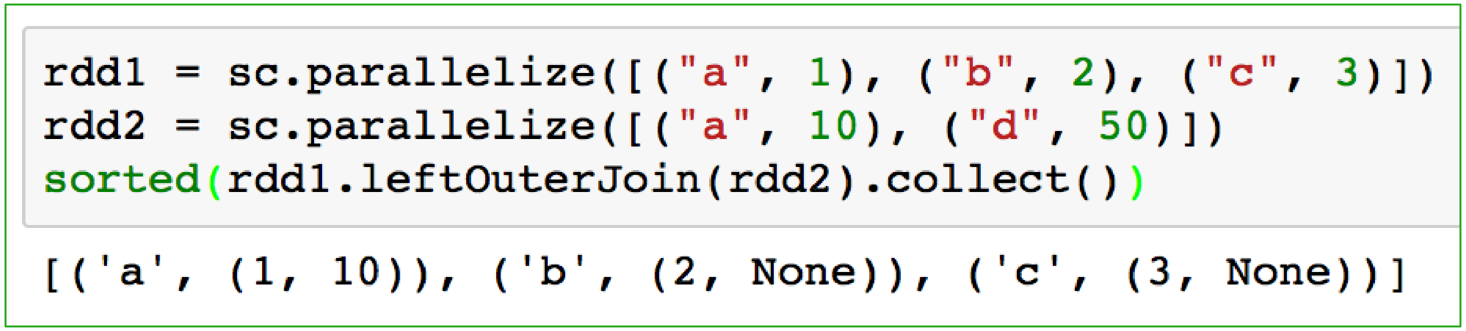
union(other RDD): Returns the union of this RDD and another one.



```
rdd1 = sc.parallelize([11, 22, 33, 44])
rdd2 = sc.parallelize([110, 220, 330, 440])
rdd1.union(rdd2).collect()

[11, 22, 33, 44, 110, 220, 330, 440]
```

Class *PySpark.RDD*

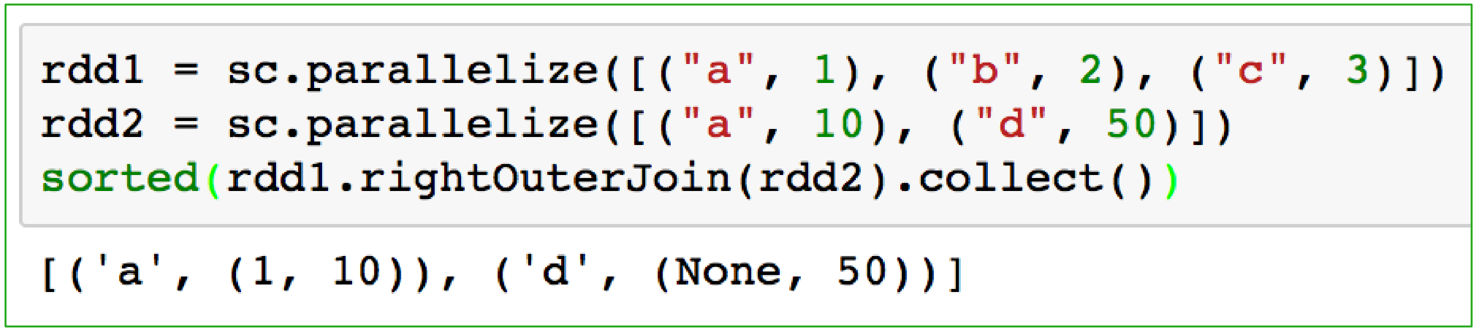
leftOuterJoin(other, *numPartitions=None*): Performs a left outer join of self and other.



```
rdd1 = sc.parallelize([("a", 1), ("b", 2), ("c", 3)])
rdd2 = sc.parallelize([("a", 10), ("d", 50)])
sorted(rdd1.leftOuterJoin(rdd2).collect())
```

```
[('a', (1, 10)), ('b', (2, None)), ('c', (3, None))]
```

rightOuterJoin(other, *numPartitions=None*): Performs a right outer join of self and other.

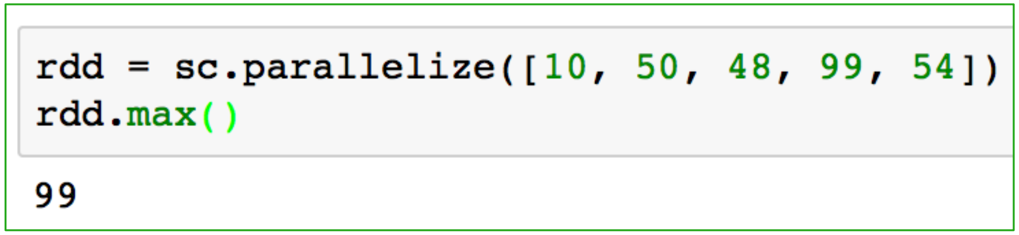


```
rdd1 = sc.parallelize([("a", 1), ("b", 2), ("c", 3)])
rdd2 = sc.parallelize([("a", 10), ("d", 50)])
sorted(rdd1.rightOuterJoin(rdd2).collect())
```

```
[('a', (1, 10)), ('d', (None, 50))]
```

Class *PySpark.RDD*

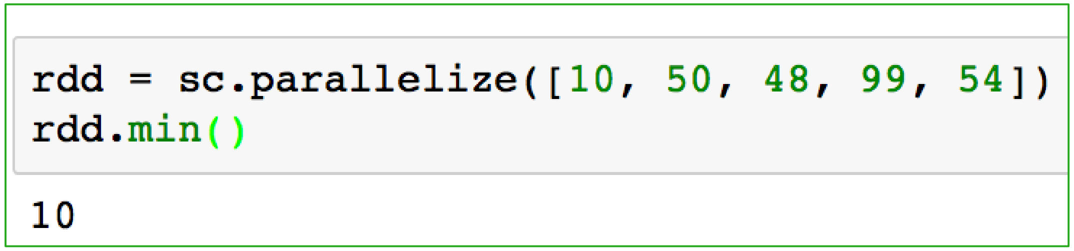
max(*key=None*): Find the maximum item in this RDD.



```
rdd = sc.parallelize([10, 50, 48, 99, 54])
rdd.max()
```

```
99
```

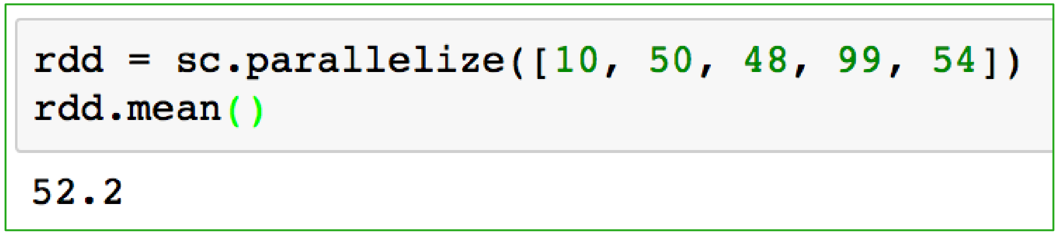
min(key=None): Finds the minimum item in this RDD.



```
rdd = sc.parallelize([10, 50, 48, 99, 54])
rdd.min()
```

```
10
```

mean(): Computes the mean of this RDD’s elements.

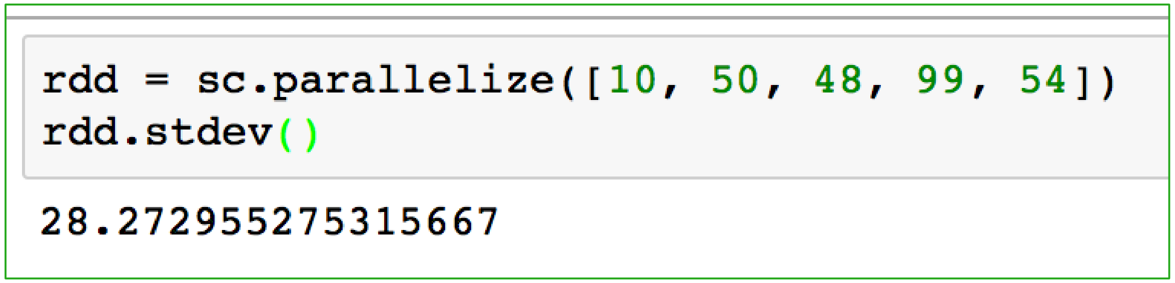


```
rdd = sc.parallelize([10, 50, 48, 99, 54])
rdd.mean()
```

```
52.2
```

Class *PySpark.RDD*

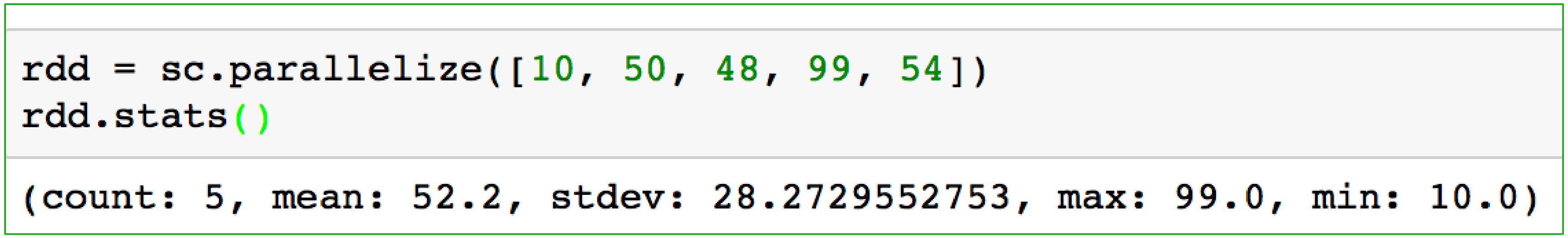
stdev(): Computes the standard deviation of this RDD’s elements.



```
rdd = sc.parallelize([10, 50, 48, 99, 54])
rdd.stdev()
```

```
28.272955275315667
```

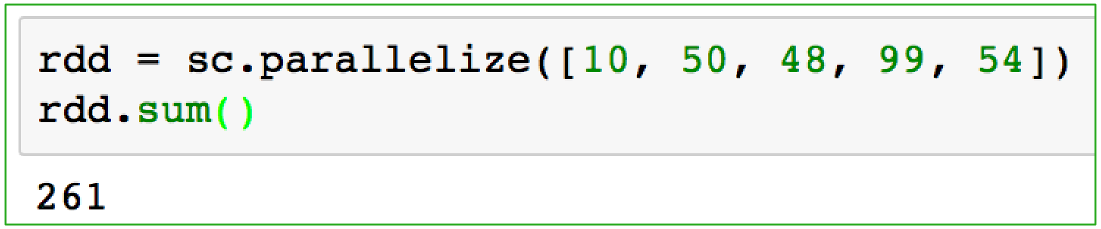
stats(): Returns a StatCounter object that captures the mean, variance and count of the RDD’s elements in one operation.



```
rdd = sc.parallelize([10, 50, 48, 99, 54])
rdd.stats()
```

```
(count: 5, mean: 52.2, stdev: 28.2729552753, max: 99.0, min: 10.0)
```

sum(): Adds up the elements in this RDD.

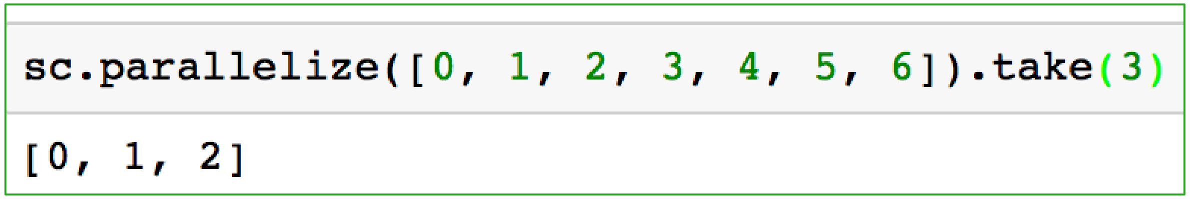


```
rdd = sc.parallelize([10, 50, 48, 99, 54])
rdd.sum()
```

```
261
```

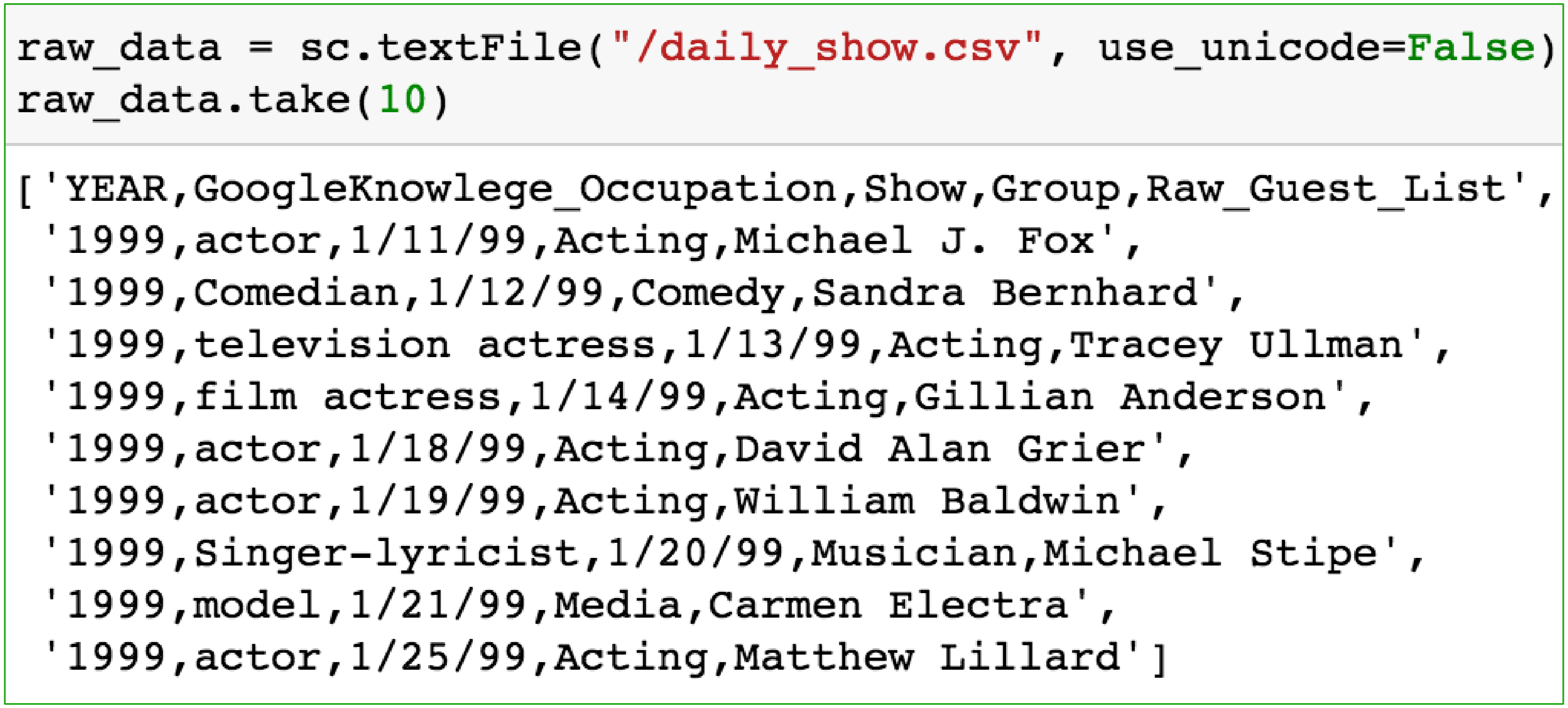
Class *PySpark.RDD*

Take(N): Takes the first N elements of the RDD as a list.



```
sc.parallelize([0, 1, 2, 3, 4, 5, 6]).take(3)
```

```
[0, 1, 2]
```

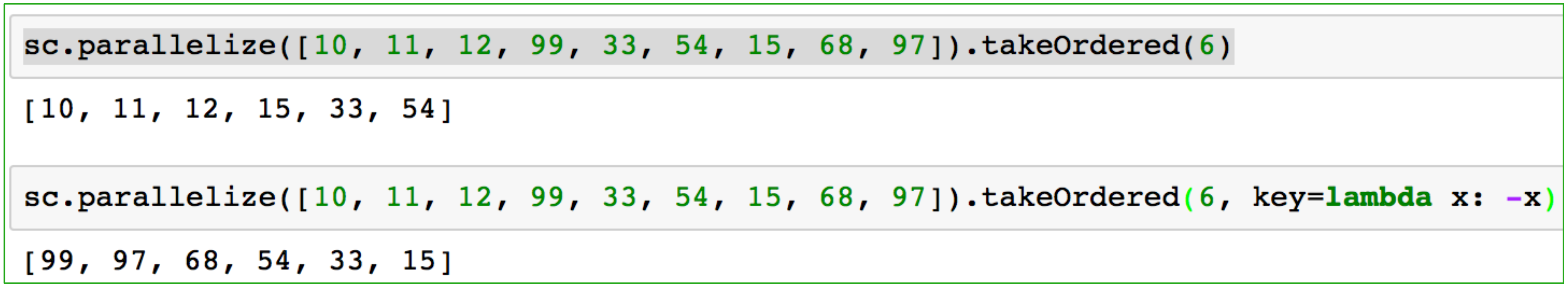


```
raw_data = sc.textFile("/daily_show.csv", use_unicode=False)
raw_data.take(10)
```

```
['YEAR,GoogleKnowlege_Occupation,Show,Group,Raw_Guest_List',
 '1999,actor,1/11/99,Acting,Michael J. Fox',
 '1999,Comedian,1/12/99,Comedy,Sandra Bernhard',
 '1999,television actress,1/13/99,Acting,Tracey Ullman',
 '1999,film actress,1/14/99,Acting,Gillian Anderson',
 '1999,actor,1/18/99,Acting,David Alan Grier',
 '1999,actor,1/19/99,Acting,William Baldwin',
 '1999,Singer-lyricist,1/20/99,Musician,Michael Stipe',
 '1999,model,1/21/99,Media,Carmen Electra',
 '1999,actor,1/25/99,Acting,Matthew Lillard']
```

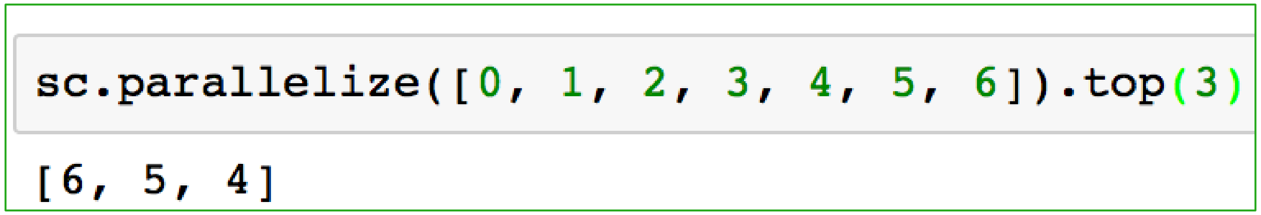
Class *PySpark.RDD*

takeOrdered(N, *key=None*): Gets the N elements from a RDD ordered in ascending order or as specified by the optional key function.



```
sc.parallelize([10, 11, 12, 99, 33, 54, 15, 68, 97]).takeOrdered(6)

[10, 11, 12, 15, 33, 54]

sc.parallelize([10, 11, 12, 99, 33, 54, 15, 68, 97]).takeOrdered(6, key=lambda x: -x)

[99, 97, 68, 54, 33, 15]
```

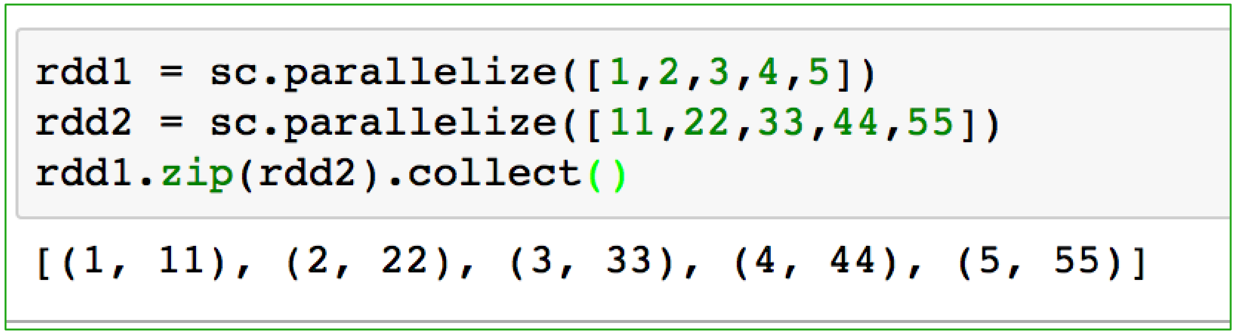
top(N, key=None): Gets the top N elements from a RDD. It returns the list sorted in descending order.



```
sc.parallelize([0, 1, 2, 3, 4, 5, 6]).top(3)

[6, 5, 4]
```

Class *PySpark.RDD*

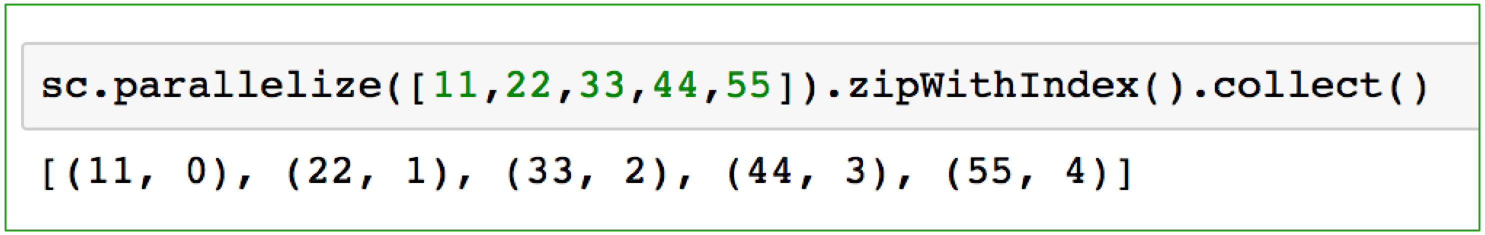
zip(other RDD): Zips this RDD with another one, returning key-value pairs with the first element in each RDD second element in each RDD, etc. Assumes that the two RDDs have the same number of partitions and the same number of elements in each partition.



```
rdd1 = sc.parallelize([1,2,3,4,5])
rdd2 = sc.parallelize([11,22,33,44,55])
rdd1.zip(rdd2).collect()
```

```
[(1, 11), (2, 22), (3, 33), (4, 44), (5, 55)]
```

zipWithIndex(): Zips this RDD with its element indices.

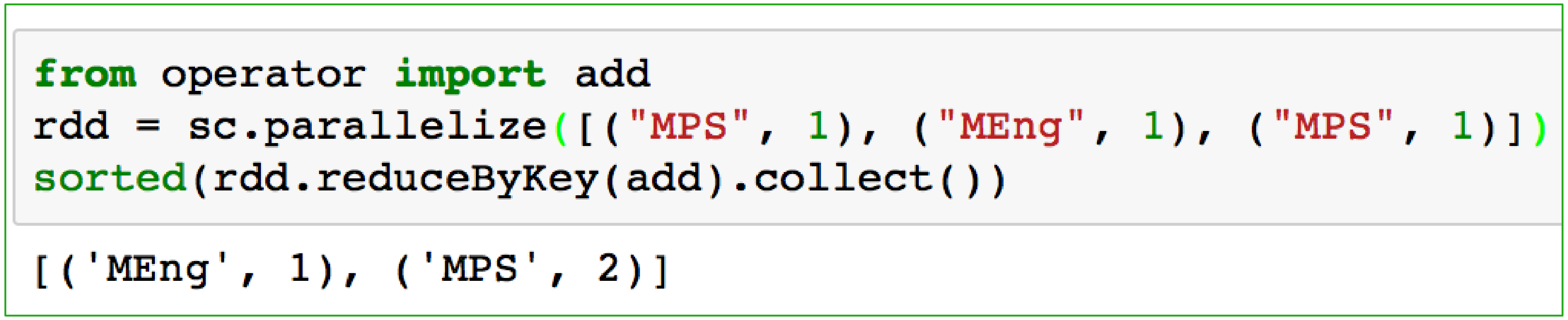


```
sc.parallelize([11,22,33,44,55]).zipWithIndex().collect()
```

```
[(11, 0), (22, 1), (33, 2), (44, 3), (55, 4)]
```

Class *PySpark.RDD*

reduceByKey(func, *numPartitions=None*): Merges the values for each key using an associative reduce function. This also performs the merging locally on each mapper before sending results to a reducer, similarly to a "combiner" in MapReduce. Output is partitioned with numPartitions partitions, or the default parallelism level if numPartitions is not specified.



```
from operator import add
rdd = sc.parallelize([("MPS", 1), ("MEng", 1), ("MPS", 1)])
sorted(rdd.reduceByKey(add).collect())
```

```
[('MEng', 1), ('MPS', 2)]
```

textFile and reduceByKey()

```python
from operator import add
lines = sc.textFile('/shakespeare.txt') #in hdfs location
counts = lines.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).reduceByKey(add)
counts.collect()
```

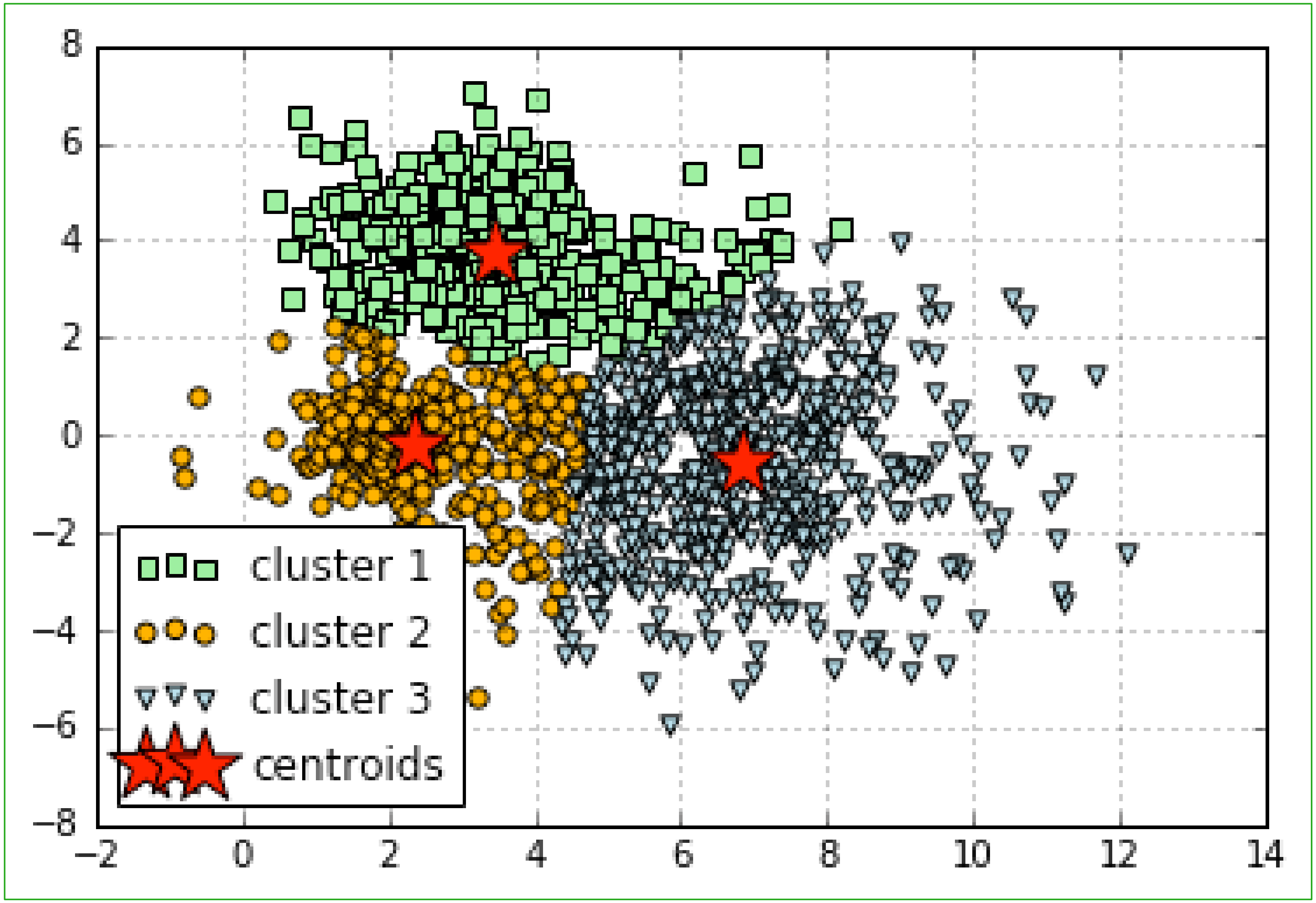
```
[(u'', 506672),
 (u'fawn', 11),
 (u'Fame,', 3),
 (u'mustachio', 1),
 (u'protested,', 1),
 (u'sending.', 3),
 (u'offendeth', 1),
 (u'instant;', 1),
 (u'scold', 4),
 (u'Sergeant.', 1),
 (u'nunnery', 1),
 (u'Sergeant,', 2),
 (u'swoopstake', 1),
 (u'unnecessarily', 1),
 (u'whither?', 3),
 (u'out-night', 1),
 (u"'Fiend,'", 1),
 (u'Retreat', 2),
 (u'Love-thoughts', 1),
 (u'spider.', 1),
 (u'spider,', 4),
 (u'chameleon.', 1),
```

K-Means Clustering

```
%pylab inline
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import KMeans

readin = sc.textFile("/unequal.txt",
                     use_unicode=False)
alldata = readin.collect()
lx=[]
ly=[]
for aline in alldata:
    splits = aline.split()
    lx.append(splits[1])
    ly.append(splits[2])
lengthList=len(lx)
x=np.array(lx).astype(float)
y=np.array(ly).astype(float)

X=np.empty((lengthList,2))
X[:,0]=x
X[:,1]=y
km = KMeans(n_clusters=3,
    init='random',
    n_init=10,
    max_iter=300,
    tol=1e-04,
    random_state=0)
y_km = km.fit_predict(X)
```

```
plt.scatter(X[y_km==0,0],
    X[y_km ==0,1],
    s=30,
    c='lightgreen',
    marker='s',
    label='cluster 1')
plt.scatter(X[y_km ==1,0],
    X[y_km ==1,1],
    s=30,
    c='orange',
    marker='o',
    label='cluster 2')
plt.scatter(X[y_km ==2,0],
    X[y_km ==2,1],
    s=30,
    c='lightblue',
     marker='v',
    label='cluster 3')
plt.scatter(km.cluster_centers_[:,0],
    km.cluster_centers_[:,1],
    s=500,
    marker='*',
    c='red',
    label='centroids')
plt.legend(loc=3)
plt.grid()
plt.show()
```

K-Means Clustering (The Plot)


cluster 1
cluster 2
cluster 3
centroids