reduce(func)

Aggregate the elements of a dataset through *func*

```
>>> names1 = sc.parallelize(["abe", "abby", "apple"])
>>> print names1.reduce(lambda t1, t2: t1+t2)
abeabbyapple

>>> names2 = sc.parallelize(["apple", "beatty", "beatrice"]).map(lambda a:
[a, len(a)])
>>> print names2.collect()
[['apple', 5], ['beatty', 6], ['beatrice', 8]]

>>> names2.flatMap(lambda t: [t[1]]).reduce(lambda t1, t2: t1+t2)
19
```

collect(func)

collect returns the elements of the RDD back to the driver program.

collect is often used in previously provided examples such as Spark Transformation Examples in Python in order to show the values of the return. Pyspark, for example, will print the values of the array back to the console. This can be helpful in debugging programs.

Examples

```
>>> sc.parallelize([1,2,3]).flatMap(lambda x: [x,x,x]).collect()
[1, 1, 1, 2, 2, 2, 3, 3, 3]
```

count()

Number of elements in the RDD

```
>>> names1 = sc.parallelize(["abe", "abby", "apple"])
>>> names1.count()
3
```

first()

Return the first element in the RDD

```
>>> names1 = sc.parallelize(["abe", "abby", "apple"])
>>> names1.first()
'abe'
```

take(n)

Take the first *n* elements of the RDD.

Works by first scanning one partition, and use the results from that partition to estimate the number of additional partitions needed to satisfy the limit.

Translated from the Scala implementation in RDD#take().

Can be much more convenient and economical to use *take* instead of *collect* to inspect a very large RDD

```
>>> names1 = sc.parallelize(["abe", "abby", "apple"])
>>> names1.take(2)
['abe', 'abby']
```

takeSample(withReplacement, n, seed=None)

Similar to *take*, in return size of n. Includes boolean option of with or without replacement and random generator seed which defaults to None

```
>>> teams = sc.parallelize(("twins", "brewers", "cubs", "white sox",
"indians", "bad news bears"))
>>> teams.takeSample(True, 3)
['brewers', 'brewers', 'twins']
# run a few times to see different results
```

countByKey()

Count the number of elements for each key, and return the result to the master as a dictionary.

```
>>> hockeyTeams = sc.parallelize(("wild", "blackhawks", "red wings",
"wild", "oilers", "whalers", "jets", "wild"))
>>> hockeyTeams.map(lambda k: (k,1)).countByKey().items()
[('red wings', 1),
 ('oilers', 1),
 ('blackhawks', 1),
 ('jets', 1),
 ('wild', 3),
 ('whalers', 1)]
```

saveAsTextFile(*path*, *compressionCodecClass=None*)

Save RDD as text file, using string representations of elements.

Parameters:	• **path** – path to file

- **compressionCodecClass** – (None by default) string i.e. “org.apache.hadoop.io.compress.GzipCodec”

```
>>> hockeyTeams = sc.parallelize(("wild", "blackhawks", "red wings",
"wild", "oilers", "whalers", "jets", "wild"))
>>> hockeyTeams.map(lambda k: (k,1)).countByKey().items()
>>> hockeyTeams.saveAsTextFile("hockey_teams.txt")
```

Produces:

```
$ ls hockey_teams.txt/
_SUCCESS       part-00001    part-00003    part-00005    part-00007
part-00000     part-00002    part-00004    part-00006
```

So, you’ll see each partition is written to it’s own file. I have 8 partitions in dataset example here.