pyspark package

Subpackages

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

Contents

PySpark is the Python API for Spark.

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`**:

  Finer-grained cache persistence levels.

class `pyspark.`**`SparkConf`**(*loadDefaults=True*, *_jvm=None*, *_jconf=None*)

Configuration for a Spark application. Used to set various Spark parameters as key-value pairs.

Most of the time, you would create a SparkConf object with **`SparkConf()`**, which will load values from *spark.** Java system properties as well. In this case, any parameters you set directly on the **`SparkConf`** object take priority over system properties.

For unit tests, you can also call **`SparkConf(false)`** to skip loading external settings and get the same configuration no matter what the system properties are.

All setter methods in this class support chaining. For example, you can write *conf.setMaster("local").setAppName("My app")*.

> **Note:** Once a SparkConf object is passed to Spark, it is cloned and can no longer be modified by the user.

`contains`(*key*)

Does this configuration contain a given key?

`get`(*key*, *defaultValue=None*)

Get the configured value for some key, or return a default otherwise.

`getAll`()

Get all values as a list of key-value pairs.

`set`(*key*, *value*)

Set a configuration property.

`setAll`(*pairs*)

Set multiple parameters, passed as a list of key-value pairs.

Parameters: **pairs** – list of key-value pairs to set

`setAppName`(*value*)

Set application name.

`setExecutorEnv`(*key=None*, *value=None*, *pairs=None*)

Set an environment variable to be passed to executors.

`setIfMissing`(*key*, *value*)

Set a configuration property, if not already set.

`setMaster`(*value*)

Set master URL to connect to.

`setSparkHome`(*value*)

Set path where Spark is installed on worker nodes.

`toDebugString`()

Returns a printable version of the configuration, as a list of key=value pairs, one per line.

class `pyspark.`**`SparkContext`**(*master=None*, *appName=None*, *sparkHome=None*, *pyFiles=None*, *environment=None*, *batchSize=0*, *serializer=PickleSerializer()*, *conf=None*, *gateway=None*, *jsc=None*, *profiler_cls=<class 'pyspark.profiler.BasicProfiler'>*)

Main entry point for Spark functionality. A SparkContext represents the connection to a Spark cluster, and can be used to create **`RDD`** and broadcast variables on that cluster.

PACKAGE_EXTENSIONS = *('.zip', '.egg', '.jar')*

accumulator(*value*, *accum_param=None*)

Create 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. Default AccumulatorParams are used for integers and floating-point numbers if you do not provide one. For other types, a custom AccumulatorParam can be used.

addFile(*path*, *recursive=False*)

Add a file to be downloaded with this Spark job on every node. The **path** passed can be either a local file, a file in HDFS (or other Hadoop-supported filesystems), or an HTTP, HTTPS or FTP URI.

To access the file in Spark jobs, use L{SparkFiles.get(fileName)<pyspark.files.SparkFiles.get>} with the filename to find its download location.

A directory can be given if the recursive option is set to True. Currently directories are only supported for Hadoop-supported filesystems.

```
>>> from pyspark import SparkFiles
>>> path = os.path.join(tempdir, "test.txt")
>>> with open(path, "w") as testFile:
...    _ = testFile.write("100")
>>> sc.addFile(path)
>>> def func(iterator):
...    with open(SparkFiles.get("test.txt")) as testFile:
...        fileVal = int(testFile.readline())
...        return [x * fileVal for x in iterator]
>>> sc.parallelize([1, 2, 3, 4]).mapPartitions(func).collect()
[100, 200, 300, 400]
```

addPyFile(*path*)

Add a .py or .zip dependency for all tasks to be executed on this SparkContext in the future. The **path** passed can be either a local file, a file in HDFS (or other Hadoop-supported filesystems), or an HTTP, HTTPS or FTP URI.

applicationId

A unique identifier for the Spark application. Its format depends on the scheduler implementation.

- in case of local spark app something like 'local-1433865536131'
- in case of YARN something like 'application_1433865536131_34483'

```
>>> sc.applicationId
u'local-...'
```

binaryFiles(*path*, *minPartitions=None*)

Note: Experimental

Read a directory of binary files from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI as a byte array. Each file is read as a single record and returned in a key-value pair, where the key is the path of each file, the value is the content of each file.

> **Note:** Small files are preferred, large file is also allowable, but may cause bad performance.

`binaryRecords`(*path*, *recordLength*)

> **Note:** Experimental

Load data from a flat binary file, assuming each record is a set of numbers with the specified numerical format (see ByteBuffer), and the number of bytes per record is constant.

Parameters:
- **path** – Directory to the input data files
- **recordLength** – The length at which to split the records

`broadcast`(*value*)

Broadcast a read-only variable to the cluster, returning a L{Broadcast<pyspark.broadcast.Broadcast>} object for reading it in distributed functions. The variable will be sent to each cluster only once.

`cancelAllJobs`()

Cancel all jobs that have been scheduled or are running.

`cancelJobGroup`(*groupId*)

Cancel active jobs for the specified group. See `SparkContext.setJobGroup` for more information.

`defaultMinPartitions`

Default min number of partitions for Hadoop RDDs when not given by user

`defaultParallelism`

Default level of parallelism to use when not given by user (e.g. for reduce tasks)

`dump_profiles`(*path*)

Dump the profile stats into directory *path*

`emptyRDD`()

Create an RDD that has no partitions or elements.

`getConf`()

`getLocalProperty`(*key*)

Get a local property set in this thread, or null if it is missing. See `setLocalProperty`

classmethod **`getOrCreate`**(*conf=None*)

Get or instantiate a SparkContext and register it as a singleton object.

Parameters: **conf** – SparkConf (optional)

`hadoopFile`(*path*, *inputFormatClass*, *keyClass*, *valueClass*, *keyConverter=None*, *valueConverter=None*, *conf=None*, *batchSize=0*)

Read an 'old' Hadoop InputFormat with arbitrary key and value class from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI. The mechanism is the same as for sc.sequenceFile.

A Hadoop configuration can be passed in as a Python dict. This will be converted into a Configuration in Java.

Parameters:

- **path** – path to Hadoop file
- **inputFormatClass** – fully qualified classname of Hadoop InputFormat (e.g. "org.apache.hadoop.mapred.TextInputFormat")
- **keyClass** – fully qualified classname of key Writable class (e.g. "org.apache.hadoop.io.Text")
- **valueClass** – fully qualified classname of value Writable class (e.g. "org.apache.hadoop.io.LongWritable")
- **keyConverter** – (None by default)
- **valueConverter** – (None by default)
- **conf** – Hadoop configuration, passed in as a dict (None by default)
- **batchSize** – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically)

`hadoopRDD`(*inputFormatClass*, *keyClass*, *valueClass*, *keyConverter=None*, *valueConverter=None*, *conf=None*, *batchSize=0*)

Read an 'old' Hadoop InputFormat with arbitrary key and value class, from an arbitrary Hadoop configuration, which is passed in as a Python dict. This will be converted into a Configuration in Java. The mechanism is the same as for sc.sequenceFile.

Parameters:

- **inputFormatClass** – fully qualified classname of Hadoop InputFormat (e.g. "org.apache.hadoop.mapred.TextInputFormat")
- **keyClass** – fully qualified classname of key Writable class (e.g. "org.apache.hadoop.io.Text")
- **valueClass** – fully qualified classname of value Writable class (e.g. "org.apache.hadoop.io.LongWritable")
- **keyConverter** – (None by default)
- **valueConverter** – (None by default)
- **conf** – Hadoop configuration, passed in as a dict (None by default)
- **batchSize** – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically)

`newAPIHadoopFile`(*path*, *inputFormatClass*, *keyClass*, *valueClass*, *keyConverter=None*, *valueConverter=None*, *conf=None*, *batchSize=0*)

Read a 'new API' Hadoop InputFormat with arbitrary key and value class from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI. The mechanism is the same as for sc.sequenceFile.

A Hadoop configuration can be passed in as a Python dict. This will be converted into a Configuration in Java

Parameters:

- **path** – path to Hadoop file
- **inputFormatClass** – fully qualified classname of Hadoop InputFormat (e.g. “org.apache.hadoop.mapreduce.lib.input.TextInputFormat”)
- **keyClass** – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.Text”)
- **valueClass** – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.LongWritable”)
- **keyConverter** – (None by default)
- **valueConverter** – (None by default)
- **conf** – Hadoop configuration, passed in as a dict (None by default)
- **batchSize** – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically)

`newAPIHadoopRDD`(*inputFormatClass*, *keyClass*, *valueClass*, *keyConverter=None*, *valueConverter=None*, *conf=None*, *batchSize=0*)

Read a ‘new API’ Hadoop InputFormat with arbitrary key and value class, from an arbitrary Hadoop configuration, which is passed in as a Python dict. This will be converted into a Configuration in Java. The mechanism is the same as for sc.sequenceFile.

Parameters:

- **inputFormatClass** – fully qualified classname of Hadoop InputFormat (e.g. “org.apache.hadoop.mapreduce.lib.input.TextInputFormat”)
- **keyClass** – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.Text”)
- **valueClass** – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.LongWritable”)
- **keyConverter** – (None by default)
- **valueConverter** – (None by default)
- **conf** – Hadoop configuration, passed in as a dict (None by default)
- **batchSize** – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically)

`parallelize`(*c*, *numSlices=None*)

Distribute a local Python collection to form an RDD. Using xrange is recommended if the input represents a range for performance.

```
>>> sc.parallelize([0, 2, 3, 4, 6], 5).glom().collect()
[[0], [2], [3], [4], [6]]
>>> sc.parallelize(xrange(0, 6, 2), 5).glom().collect()
[[], [0], [], [2], [4]]
```

`pickleFile`(*name*, *minPartitions=None*)

Load an RDD previously saved using **`RDD.saveAsPickleFile`** method.

```
>>> tmpFile = NamedTemporaryFile(delete=True)
>>> tmpFile.close()
>>> sc.parallelize(range(10)).saveAsPickleFile(tmpFile.name, 5)
>>> sorted(sc.pickleFile(tmpFile.name, 3).collect())
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

range(*start*, *end=None*, *step=1*, *numSlices=None*)

Create a new RDD of int containing elements from *start* to *end* (exclusive), increased by *step* every element. Can be called the same way as python’s built-in range() function. If called with a single argument, the argument is interpreted as *end*, and *start* is set to 0.

Parameters:

- **start** – the start value
- **end** – the end value (exclusive)
- **step** – the incremental step (default: 1)
- **numSlices** – the number of partitions of the new RDD

Returns: An RDD of int

```
>>> sc.range(5).collect()
[0, 1, 2, 3, 4]
>>> sc.range(2, 4).collect()
[2, 3]
>>> sc.range(1, 7, 2).collect()
[1, 3, 5]
```

runJob(*rdd*, *partitionFunc*, *partitions=None*, *allowLocal=False*)

Executes the given partitionFunc on the specified set of partitions, returning the result as an array of elements.

If ‘partitions’ is not specified, this will run over all partitions.

```
>>> myRDD = sc.parallelize(range(6), 3)
>>> sc.runJob(myRDD, lambda part: [x * x for x in part])
[0, 1, 4, 9, 16, 25]
```

```
>>> myRDD = sc.parallelize(range(6), 3)
>>> sc.runJob(myRDD, lambda part: [x * x for x in part], [0, 2], True)
[0, 1, 16, 25]
```

sequenceFile(*path*, *keyClass=None*, *valueClass=None*, *keyConverter=None*, *valueConverter=None*, *minSplits=None*, *batchSize=0*)

Read a Hadoop SequenceFile with arbitrary key and value Writable class from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI. The mechanism is as follows:

1. A Java RDD is created from the SequenceFile or other InputFormat, and the key and value Writable classes
2. Serialization is attempted via Pyrolite pickling

3. If this fails, the fallback is to call ‘toString’ on each key and value
4. `PickleSerializer` is used to deserialize pickled objects on the Python side

Parameters:
- **path** – path to sequncefile
- **keyClass** – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.Text”)
- **valueClass** – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.LongWritable”)
- **keyConverter** –
- **valueConverter** –
- **minSplits** – minimum splits in dataset (default min(2, sc.defaultParallelism))
- **batchSize** – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically)

`setCheckpointDir`(*dirName*)

Set the directory under which RDDs are going to be checkpointed. The directory must be a HDFS path if running on a cluster.

`setJobGroup`(*groupId*, *description*, *interruptOnCancel=False*)

Assigns a group ID to all the jobs started by this thread until the group ID is set to a different value or cleared.

Often, a unit of execution in an application consists of multiple Spark actions or jobs. Application programmers can use this method to group all those jobs together and give a group description. Once set, the Spark web UI will associate such jobs with this group.

The application can use `SparkContext.cancelJobGroup` to cancel all running jobs in this group.

```
>>> import threading
>>> from time import sleep
>>> result = "Not Set"
>>> lock = threading.Lock()
>>> def map_func(x):
...     sleep(100)
...     raise Exception("Task should have been cancelled")
>>> def start_job(x):
...     global result
...     try:
...         sc.setJobGroup("job_to_cancel", "some description")
...         result = sc.parallelize(range(x)).map(map_func).collect()
...     except Exception as e:
...         result = "Cancelled"
...     lock.release()
>>> def stop_job():
...     sleep(5)
...     sc.cancelJobGroup("job_to_cancel")
>>> supress = lock.acquire()
>>> supress = threading.Thread(target=start_job, args=(10,)).start()
>>> supress = threading.Thread(target=stop_job).start()
>>> supress = lock.acquire()
>>> print(result)
Cancelled
```

If interruptOnCancel is set to true for the job group, then job cancellation will result in Thread.interrupt() being called on the job's executor threads. This is useful to help ensure that the tasks are actually stopped in a timely manner, but is off by default due to HDFS-1208, where HDFS may respond to Thread.interrupt() by marking nodes as dead.

setLocalProperty(*key*, *value*)

Set a local property that affects jobs submitted from this thread, such as the Spark fair scheduler pool.

setLogLevel(*logLevel*)

Control our logLevel. This overrides any user-defined log settings. Valid log levels include: ALL, DEBUG, ERROR, FATAL, INFO, OFF, TRACE, WARN

classmethod **setSystemProperty**(*key*, *value*)

Set a Java system property, such as spark.executor.memory. This must must be invoked before instantiating SparkContext.

show_profiles()

Print the profile stats to stdout

sparkUser()

Get SPARK_USER for user who is running SparkContext.

startTime

Return the epoch time when the Spark Context was started.

statusTracker()

Return `StatusTracker` object

stop()

Shut down the SparkContext.

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

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

If use_unicode is False, the strings will be kept as *str* (encoding as *utf-8*), which is faster and smaller than unicode. (Added in Spark 1.2)

```
>>> path = os.path.join(tempdir, "sample-text.txt")
>>> with open(path, "w") as testFile:
...    _ = testFile.write("Hello world!")
>>> textFile = sc.textFile(path)
>>> textFile.collect()
[u'Hello world!']
```

uiWebUrl

Return the URL of the SparkUI instance started by this SparkContext

union(*rdds*)

Build the union of a list of RDDs.

This supports unions() of RDDs with different serialized formats, although this forces them to be reserialized using the default serializer:

```
>>> path = os.path.join(tempdir, "union-text.txt")
>>> with open(path, "w") as testFile:
...     _ = testFile.write("Hello")
>>> textFile = sc.textFile(path)
>>> textFile.collect()
[u'Hello']
>>> parallelized = sc.parallelize(["World!"])
>>> sorted(sc.union([textFile, parallelized]).collect())
[u'Hello', 'World!']
```

version

The version of Spark on which this application is running.

wholeTextFiles(*path*, *minPartitions=None*, *use_unicode=True*)

Read a directory of text files from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI. Each file is read as a single record and returned in a key-value pair, where the key is the path of each file, the value is the content of each file.

If use_unicode is False, the strings will be kept as *str* (encoding as *utf-8*), which is faster and smaller than unicode. (Added in Spark 1.2)

For example, if you have the following files:

```
hdfs://a-hdfs-path/part-00000
hdfs://a-hdfs-path/part-00001
...
hdfs://a-hdfs-path/part-nnnnn
```

Do *rdd = sparkContext.wholeTextFiles("hdfs://a-hdfs-path")*, then `rdd` contains:

```
(a-hdfs-path/part-00000, its content)
(a-hdfs-path/part-00001, its content)
...
(a-hdfs-path/part-nnnnn, its content)
```

Note: Small files are preferred, as each file will be loaded fully in memory.

```
>>> dirPath = os.path.join(tempdir, "files")
>>> os.mkdir(dirPath)
>>> with open(os.path.join(dirPath, "1.txt"), "w") as file1:
...     _ = file1.write("1")
>>> with open(os.path.join(dirPath, "2.txt"), "w") as file2:
...     _ = file2.write("2")
>>> textFiles = sc.wholeTextFiles(dirPath)
>>> sorted(textFiles.collect())
[(u'.../1.txt', u'1'), (u'.../2.txt', u'2')]
```

class `pyspark.`**`SparkFiles`**

Resolves paths to files added through L{SparkContext.addFile() <pyspark.context.SparkContext.addFile>}.

SparkFiles contains only classmethods; users should not create SparkFiles instances.

classmethod **`get`**(*filename*)

Get the absolute path of a file added through **`SparkContext.addFile()`**.

classmethod **`getRootDirectory`**()

Get the root directory that contains files added through **`SparkContext.addFile()`**.

class `pyspark.`**`RDD`**(*jrdd*, *ctx*, *jrdd_deserializer=AutoBatchedSerializer(PickleSerializer())*)

A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable, partitioned collection of elements that can be operated on in parallel.

`aggregate`(*zeroValue*, *seqOp*, *combOp*)

Aggregate the elements of each partition, and then the results for all the partitions, using a given combine functions and a neutral “zero value.”

The functions *op(t1, t2)* is allowed to modify **`t1`** and return it as its result value to avoid object allocation; however, it should not modify **`t2`**.

The first function (seqOp) can return a different result type, U, than the type of this RDD. Thus, we need one operation for merging a T into an U and one operation for merging two U

```
>>> seqOp = (lambda x, y: (x[0] + y, x[1] + 1))
>>> combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))
>>> sc.parallelize([1, 2, 3, 4]).aggregate((0, 0), seqOp, combOp)
(10, 4)
>>> sc.parallelize([]).aggregate((0, 0), seqOp, combOp)
(0, 0)
```

`aggregateByKey`(*zeroValue*, *seqFunc*, *combFunc*, *numPartitions=None*, *partitionFunc=<function portable_hash at 0x7fc35dbc8e60>*)

Aggregate the values of each key, using given combine functions and a neutral “zero value”. This function can return a different result type, U, than the type of the values in this RDD, V. Thus, we need one operation for merging a V into a U and one operation for merging two U’s, The former operation is used for merging values within a partition, and the latter is used for merging values between partitions. To avoid memory allocation, both of these functions are allowed to modify and return their first argument instead of creating a new U.

`cache`()

Persist this RDD with the default storage level (**`MEMORY_ONLY`**).

`cartesian`(*other*)

Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements *(a, b)* where `a` is in `self` and `b` is in `other`.

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

`checkpoint`()

Mark this RDD for checkpointing. It will be saved to a file inside the checkpoint directory set with `SparkContext.setCheckpointDir()` and all references to its parent RDDs will be removed. This function must be called before any job has been executed on this RDD. It is strongly recommended that this RDD is persisted in memory, otherwise saving it on a file will require recomputation.

`coalesce`(*numPartitions*, *shuffle=False*)

Return a new RDD that is reduced into *numPartitions* partitions.

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

`cogroup`(*other*, *numPartitions=None*)

For each key k in `self` or `other`, return 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", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> [(x, tuple(map(list, y))) for x, y in sorted(list(x.cogroup(y).collect()))]
[('a', ([1], [2])), ('b', ([4], []))]
```

`collect`()

Return a list that contains all of the elements in this RDD.

> **Note:** This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver’s memory.

`collectAsMap`()

Return the key-value pairs in this RDD to the master as a dictionary.

> **Note:** this method should only be used if the resulting data is expected to be small, as all the data is loaded into the driver’s memory.

```
>>> m = sc.parallelize([(1, 2), (3, 4)]).collectAsMap()
>>> m[1]
2
>>> m[3]
4
```

`combineByKey`(*createCombiner*, *mergeValue*, *mergeCombiners*, *numPartitions=None*, *partitionFunc=<function portable_hash at 0x7fc35dbc8e60>*)

Generic function to combine the elements for each key using a custom set of aggregation functions.

Turns an RDD[(K, V)] into a result of type RDD[(K, C)], for a “combined type” C.

Users provide three functions:

- **`createCombiner`**, which turns a V into a C (e.g., creates a one-element list)
- **`mergeValue`**, to merge a V into a C (e.g., adds it to the end of a list)
- **`mergeCombiners`**, to combine two C’s into a single one.

In addition, users can control the partitioning of the output RDD.

Note: V and C can be different – for example, one might group an RDD of type (Int, Int) into an RDD of type (Int, List[Int]).

```
>>> x = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> def add(a, b): return a + str(b)
>>> sorted(x.combineByKey(str, add, add).collect())
[('a', '11'), ('b', '1')]
```

`context`

The **`SparkContext`** that this RDD was created on.

`count`()

Return the number of elements in this RDD.

```
>>> sc.parallelize([2, 3, 4]).count()
3
```

`countApprox`(*timeout*, *confidence=0.95*)

Note: Experimental

Approximate version of count() that returns a potentially incomplete result within a timeout, even if not all tasks have finished.

```
>>> rdd = sc.parallelize(range(1000), 10)
>>> rdd.countApprox(1000, 1.0)
1000
```

countApproxDistinct(*relativeSD=0.05*)

Note: Experimental

Return approximate number of distinct elements in the RDD.

The algorithm used is based on streamlib's implementation of "HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm", available here.

Parameters: **relativeSD** – Relative accuracy. Smaller values create counters that require more space. It must be greater than 0.000017.

```
>>> n = sc.parallelize(range(1000)).map(str).countApproxDistinct()
>>> 900 < n < 1100
True
>>> n = sc.parallelize([i % 20 for i in range(1000)]).countApproxDistinct()
>>> 16 < n < 24
True
```

countByKey()

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

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.countByKey().items())
[('a', 2), ('b', 1)]
```

countByValue()

Return the count of each unique value in this RDD as a dictionary of (value, count) pairs.

```
>>> sorted(sc.parallelize([1, 2, 1, 2, 2], 2).countByValue().items())
[(1, 2), (2, 3)]
```

distinct(*numPartitions=None*)

Return a new RDD containing the distinct elements in this RDD.

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

filter(*f*)

Return a new RDD containing only the elements that satisfy a predicate.

```
>>> rdd = sc.parallelize([1, 2, 3, 4, 5])
>>> rdd.filter(lambda x: x % 2 == 0).collect()
[2, 4]
```

first()

Return the first element in this RDD.

```
>>> sc.parallelize([2, 3, 4]).first()
2
>>> sc.parallelize([]).first()
Traceback (most recent call last):
    ...
ValueError: RDD is empty
```

flatMap(*f*, *preservesPartitioning=False*)

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

```
>>> rdd = sc.parallelize([2, 3, 4])
>>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())
[1, 1, 1, 2, 2, 3]
>>> sorted(rdd.flatMap(lambda x: [(x, x), (x, x)]).collect())
[(2, 2), (2, 2), (3, 3), (3, 3), (4, 4), (4, 4)]
```

flatMapValues(*f*)

Pass each value in the key-value pair RDD through a flatMap function without changing the keys; this also retains the original RDD's partitioning.

```
>>> x = sc.parallelize([("a", ["x", "y", "z"]), ("b", ["p", "r"])])
>>> def f(x): return x
>>> x.flatMapValues(f).collect()
[('a', 'x'), ('a', 'y'), ('a', 'z'), ('b', 'p'), ('b', 'r')]
```

fold(*zeroValue*, *op*)

Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral "zero value."

The function *op(t1, t2)* is allowed to modify `t1` and return it as its result value to avoid object allocation; however, it should not modify `t2`.

This behaves somewhat differently from fold operations implemented for non-distributed collections in functional languages like Scala. This fold operation may be applied to partitions individually, and then fold those results into the final result, rather than apply the fold to each element sequentially in some defined ordering. For functions that are not commutative, the result may differ from that of a fold applied to a non-distributed collection.

```
>>> from operator import add
>>> sc.parallelize([1, 2, 3, 4, 5]).fold(0, add)
15
```

foldByKey(*zeroValue*, *func*, *numPartitions=None*, *partitionFunc=<function portable_hash at 0x7fc35dbc8e60>*)

Merge the values for each key using an associative function “func” and a neutral “zeroValue” which may be added to the result an arbitrary number of times, and must not change the result (e.g., 0 for addition, or 1 for multiplication.).

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> from operator import add
>>> sorted(rdd.foldByKey(0, add).collect())
[('a', 2), ('b', 1)]
```

foreach(*f*)

Applies a function to all elements of this RDD.

```
>>> def f(x): print(x)
>>> sc.parallelize([1, 2, 3, 4, 5]).foreach(f)
```

foreachPartition(*f*)

Applies a function to each partition of this RDD.

```
>>> def f(iterator):
...     for x in iterator:
...          print(x)
>>> sc.parallelize([1, 2, 3, 4, 5]).foreachPartition(f)
```

fullOuterJoin(*other*, *numPartitions=None*)

Perform a right outer join of **`self`** and **`other`**.

For each element (k, v) in **`self`**, the resulting RDD will either contain all pairs (k, (v, w)) for w in **`other`**, or the pair (k, (v, None)) if no elements in **`other`** have key k.

Similarly, for each element (k, w) in **`other`**, the resulting RDD will either contain all pairs (k, (v, w)) for v in **`self`**, or the pair (k, (None, w)) if no elements in **`self`** have key k.

Hash-partitions the resulting RDD into the given number of partitions.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("c", 8)])
>>> sorted(x.fullOuterJoin(y).collect())
[('a', (1, 2)), ('b', (4, None)), ('c', (None, 8))]
```

getCheckpointFile()

Gets the name of the file to which this RDD was checkpointed

Not defined if RDD is checkpointed locally.

getNumPartitions()

Returns the number of partitions in RDD

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

`getStorageLevel`()

Get the RDD’s current storage level.

```
>>> rdd1 = sc.parallelize([1,2])
>>> rdd1.getStorageLevel()
StorageLevel(False, False, False, False, 1)
>>> print(rdd1.getStorageLevel())
Serialized 1x Replicated
```

`glom`()

Return 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]]
```

`groupBy`(*f*, *numPartitions=None*, *partitionFunc=<function portable_hash at 0x7fc35dbc8e60>*)

Return an RDD of grouped items.

```
>>> rdd = sc.parallelize([1, 1, 2, 3, 5, 8])
>>> result = rdd.groupBy(lambda x: x % 2).collect()
>>> sorted([(x, sorted(y)) for (x, y) in result])
[(0, [2, 8]), (1, [1, 1, 3, 5])]
```

`groupByKey`(*numPartitions=None*, *partitionFunc=<function portable_hash at 0x7fc35dbc8e60>*)

Group the values for each key in the RDD into a single sequence. Hash-partitions the resulting RDD with numPartitions partitions.

Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will provide much better performance.

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.groupByKey().mapValues(len).collect())
[('a', 2), ('b', 1)]
>>> sorted(rdd.groupByKey().mapValues(list).collect())
[('a', [1, 1]), ('b', [1])]
```

`groupWith`(*other*, **others*)

Alias for cogroup but with support for multiple RDDs.

```
>>> w = sc.parallelize([("a", 5), ("b", 6)])
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> z = sc.parallelize([("b", 42)])
>>> [(x, tuple(map(list, y))) for x, y in sorted(list(w.groupWith(x, y, z).collect()))
[('a', ([5], [1], [2], [])), ('b', ([6], [4], [], [42]))]
```

histogram(*buckets*)

Compute a histogram using the provided buckets. The buckets 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], which means 1<=x<10, 10<=x<20, 20<=x<=50. And on the input of 1 and 50 we would have a histogram of 1,0,1.

If your histogram is evenly spaced (e.g. [0, 10, 20, 30]), this can be switched from an O(log n) inseration to O(1) per element (where n is the number of buckets).

Buckets must be sorted, not contain any duplicates, and have at least two elements.

If *buckets* is a number, it will generate buckets which are evenly spaced between the minimum and maximum of the RDD. For example, if the min value is 0 and the max is 100, given *buckets* as 2, the resulting buckets will be [0,50) [50,100]. *buckets* must be at least 1. An exception is raised if the RDD contains infinity. If the elements in the RDD do not vary (max == min), a single bucket will be used.

The return value is a tuple of buckets and histogram.

```
>>> rdd = sc.parallelize(range(51))
>>> rdd.histogram(2)
([0, 25, 50], [25, 26])
>>> rdd.histogram([0, 5, 25, 50])
([0, 5, 25, 50], [5, 20, 26])
>>> rdd.histogram([0, 15, 30, 45, 60])  # evenly spaced buckets
([0, 15, 30, 45, 60], [15, 15, 15, 6])
>>> rdd = sc.parallelize(["ab", "ac", "b", "bd", "ef"])
>>> rdd.histogram(("a", "b", "c"))
(('a', 'b', 'c'), [2, 2])
```

id()

A unique ID for this RDD (within its SparkContext).

intersection(*other*)

Return the intersection of this RDD and another one. The output will not contain any duplicate elements, even if the input RDDs did.

Note: This method performs a shuffle internally.

```
>>> rdd1 = sc.parallelize([1, 10, 2, 3, 4, 5])
>>> rdd2 = sc.parallelize([1, 6, 2, 3, 7, 8])
>>> rdd1.intersection(rdd2).collect()
[1, 2, 3]
```

isCheckpointed()

Return whether this RDD is checkpointed and materialized, either reliably or locally.

isEmpty()

Returns true if and only if the RDD contains no elements at all.

Note: an RDD may be empty even when it has at least 1 partition.

```
>>> sc.parallelize([]).isEmpty()
True
>>> sc.parallelize([1]).isEmpty()
False
```

isLocallyCheckpointed()

Return whether this RDD is marked for local checkpointing.

Exposed for testing.

join(*other*, *numPartitions=None*)

Return an RDD containing all pairs of elements with matching keys in **self** and **other**.

Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in **self** and (k, v2) is in **other**.

Performs a hash join across the cluster.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> sorted(x.join(y).collect())
[('a', (1, 2)), ('a', (1, 3))]
```

keyBy(*f*)

Creates tuples of the elements in this RDD by applying **f**.

```
>>> x = sc.parallelize(range(0,3)).keyBy(lambda x: x*x)
>>> y = sc.parallelize(zip(range(0,5), range(0,5)))
>>> [(x, list(map(list, y))) for x, y in sorted(x.cogroup(y).collect())]
[(0, [[0], [0]]), (1, [[1], [1]]), (2, [[], [2]]), (3, [[], [3]]), (4, [[2], [4]])]
```

keys()

Return an RDD with the keys of each tuple.

```
>>> m = sc.parallelize([(1, 2), (3, 4)]).keys()
>>> m.collect()
```

```
[1, 3]
```

`leftOuterJoin`(*other*, *numPartitions=None*)

Perform a left outer join of **`self`** and **`other`**.

For each element (k, v) in **`self`**, the resulting RDD will either contain all pairs (k, (v, w)) for w in **`other`**, or the pair (k, (v, None)) if no elements in **`other`** have key k.

Hash-partitions the resulting RDD into the given number of partitions.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> sorted(x.leftOuterJoin(y).collect())
[('a', (1, 2)), ('b', (4, None))]
```

`localCheckpoint`()

Mark this RDD for local checkpointing using Spark's existing caching layer.

This method is for users who wish to truncate RDD lineages while skipping the expensive step of replicating the materialized data in a reliable distributed file system. This is useful for RDDs with long lineages that need to be truncated periodically (e.g. GraphX).

Local checkpointing sacrifices fault-tolerance for performance. In particular, checkpointed data is written to ephemeral local storage in the executors instead of to a reliable, fault-tolerant storage. The effect is that if an executor fails during the computation, the checkpointed data may no longer be accessible, causing an irrecoverable job failure.

This is NOT safe to use with dynamic allocation, which removes executors along with their cached blocks. If you must use both features, you are advised to set **`spark.dynamicAllocation.cachedExecutorIdleTimeout`** to a high value.

The checkpoint directory set through **`SparkContext.setCheckpointDir()`** is not used.

`lookup`(*key*)

Return the list of values in the RDD for key *key*. This operation is done efficiently if the RDD has a known partitioner by only searching the partition that the key maps to.

```
>>> l = range(1000)
>>> rdd = sc.parallelize(zip(l, l), 10)
>>> rdd.lookup(42)  # slow
[42]
>>> sorted = rdd.sortByKey()
>>> sorted.lookup(42)  # fast
[42]
>>> sorted.lookup(1024)
[]
>>> rdd2 = sc.parallelize([(('a', 'b'), 'c')]).groupByKey()
>>> list(rdd2.lookup(('a', 'b'))[0])
['c']
```

map(*f*, *preservesPartitioning=False*)

Return a new RDD by applying a function to each element of this RDD.

```
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> sorted(rdd.map(lambda x: (x, 1)).collect())
[('a', 1), ('b', 1), ('c', 1)]
```

mapPartitions(*f*, *preservesPartitioning=False*)

Return a new RDD by applying a function to each partition of this RDD.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> def f(iterator): yield sum(iterator)
>>> rdd.mapPartitions(f).collect()
[3, 7]
```

mapPartitionsWithIndex(*f*, *preservesPartitioning=False*)

Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 4)
>>> def f(splitIndex, iterator): yield splitIndex
>>> rdd.mapPartitionsWithIndex(f).sum()
6
```

mapPartitionsWithSplit(*f*, *preservesPartitioning=False*)

Deprecated: use mapPartitionsWithIndex instead.

Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 4)
>>> def f(splitIndex, iterator): yield splitIndex
>>> rdd.mapPartitionsWithSplit(f).sum()
6
```

mapValues(*f*)

Pass each value in the key-value pair RDD through a map function without changing the keys; this also retains the original RDD’s partitioning.

```
>>> x = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b", ["grapes"])])
>>> def f(x): return len(x)
>>> x.mapValues(f).collect()
[('a', 3), ('b', 1)]
```

max(*key=None*)

Find the maximum item in this RDD.

Parameters: **key** – A function used to generate key for comparing

```
>>> rdd = sc.parallelize([1.0, 5.0, 43.0, 10.0])
>>> rdd.max()
43.0
>>> rdd.max(key=str)
5.0
```

mean()

Compute the mean of this RDD's elements.

```
>>> sc.parallelize([1, 2, 3]).mean()
2.0
```

meanApprox(*timeout*, *confidence=0.95*)

Note: Experimental

Approximate operation to return the mean within a timeout or meet the confidence.

```
>>> rdd = sc.parallelize(range(1000), 10)
>>> r = sum(range(1000)) / 1000.0
>>> abs(rdd.meanApprox(1000) - r) / r < 0.05
True
```

min(*key=None*)

Find the minimum item in this RDD.

Parameters: **key** – A function used to generate key for comparing

```
>>> rdd = sc.parallelize([2.0, 5.0, 43.0, 10.0])
>>> rdd.min()
2.0
>>> rdd.min(key=str)
10.0
```

name()

Return the name of this RDD.

partitionBy(*numPartitions*, *partitionFunc=<function portable_hash at 0x7fc35dbc8e60>*)

Return a copy of the RDD partitioned using the specified partitioner.

```
>>> pairs = sc.parallelize([1, 2, 3, 4, 2, 4, 1]).map(lambda x: (x, x))
>>> sets = pairs.partitionBy(2).glom().collect()
>>> len(set(sets[0]).intersection(set(sets[1])))
0
```

persist(*storageLevel=StorageLevel(False, True, False, False, 1)*)

Set this RDD’s storage level to persist its values across operations after the first time it is computed. This can only be used to assign a new storage level if the RDD does not have a storage level set yet. If no storage level is specified defaults to (`MEMORY_ONLY`).

```
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> rdd.persist().is_cached
True
```

pipe(*command*, *env=None*, *checkCode=False*)

Return an RDD created by piping elements to a forked external process.

```
>>> sc.parallelize(['1', '2', '', '3']).pipe('cat').collect()
[u'1', u'2', u'', u'3']
```

Parameters: **checkCode** – whether or not to check the return value of the shell command.

randomSplit(*weights*, *seed=None*)

Randomly splits this RDD with the provided weights.

Parameters:
- **weights** – weights for splits, will be normalized if they don’t sum to 1
- **seed** – random seed

Returns: split RDDs in a list

```
>>> rdd = sc.parallelize(range(500), 1)
>>> rdd1, rdd2 = rdd.randomSplit([2, 3], 17)
>>> len(rdd1.collect() + rdd2.collect())
500
>>> 150 < rdd1.count() < 250
True
>>> 250 < rdd2.count() < 350
True
```

reduce(*f*)

Reduces the elements of this RDD using the specified commutative and associative binary operator. Currently reduces partitions locally.

```
>>> from operator import add
>>> sc.parallelize([1, 2, 3, 4, 5]).reduce(add)
15
>>> sc.parallelize((2 for _ in range(10))).map(lambda x: 1).cache().reduce(add)
10
>>> sc.parallelize([]).reduce(add)
Traceback (most recent call last):
    ...
ValueError: Can not reduce() empty RDD
```

reduceByKey(*func*, *numPartitions=None*, *partitionFunc=<function portable_hash at 0x7fc35dbc8e60>*)

Merge the values for each key using an associative and commutative reduce function.

This will also perform the merging locally on each mapper before sending results to a reducer, similarly to a “combiner” in MapReduce.

Output will be partitioned with `numPartitions` partitions, or the default parallelism level if `numPartitions` is not specified. Default partitioner is hash-partition.

```
>>> from operator import add
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.reduceByKey(add).collect())
[('a', 2), ('b', 1)]
```

reduceByKeyLocally(*func*)

Merge the values for each key using an associative and commutative reduce function, but return the results immediately to the master as a dictionary.

This will also perform the merging locally on each mapper before sending results to a reducer, similarly to a “combiner” in MapReduce.

```
>>> from operator import add
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.reduceByKeyLocally(add).items())
[('a', 2), ('b', 1)]
```

repartition(*numPartitions*)

Return a new RDD that has exactly numPartitions partitions.

Can increase or decrease the level of parallelism in this RDD. Internally, this uses a shuffle to redistribute data. If you are decreasing the number of partitions in this RDD, consider using *coalesce*, which can avoid performing a shuffle.

```
>>> rdd = sc.parallelize([1,2,3,4,5,6,7], 4)
>>> sorted(rdd.glom().collect())
[[1], [2, 3], [4, 5], [6, 7]]
>>> len(rdd.repartition(2).glom().collect())
2
>>> len(rdd.repartition(10).glom().collect())
10
```

repartitionAndSortWithinPartitions(*numPartitions=None*, *partitionFunc=<function portable_hash at 0x7fc35dbc8e60>*, *ascending=True*, *keyfunc=<function <lambda> at 0x7fc35dbcf758>*)

Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys.

```
>>> rdd = sc.parallelize([(0, 5), (3, 8), (2, 6), (0, 8), (3, 8), (1, 3)])
>>> rdd2 = rdd.repartitionAndSortWithinPartitions(2, lambda x: x % 2, 2)
>>> rdd2.glom().collect()
[[(0, 5), (0, 8), (2, 6)], [(1, 3), (3, 8), (3, 8)]]
```

rightOuterJoin(*other*, *numPartitions=None*)

Perform a right outer join of **self** and **other**.

For each element (k, w) in **other**, the resulting RDD will either contain all pairs (k, (v, w)) for v in this, or the pair (k, (None, w)) if no elements in **self** have key k.

Hash-partitions the resulting RDD into the given number of partitions.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> sorted(y.rightOuterJoin(x).collect())
[('a', (2, 1)), ('b', (None, 4))]
```

sample(*withReplacement*, *fraction*, *seed=None*)

Return a sampled subset of this RDD.

Parameters:

- **withReplacement** – can elements be sampled multiple times (replaced when sampled out)
- **fraction** – expected size of the sample as a fraction of this RDD's size without replacement: probability that each element is chosen; fraction must be [0, 1] with replacement: expected number of times each element is chosen; fraction must be >= 0
- **seed** – seed for the random number generator

Note: This is not guaranteed to provide exactly the fraction specified of the total count of the given **DataFrame**.

```
>>> rdd = sc.parallelize(range(100), 4)
>>> 6 <= rdd.sample(False, 0.1, 81).count() <= 14
True
```

sampleByKey(*withReplacement*, *fractions*, *seed=None*)

Return a subset of this RDD sampled by key (via stratified sampling). Create a sample of this RDD using variable sampling rates for different keys as specified by fractions, a key to sampling rate map.

```
>>> fractions = {"a": 0.2, "b": 0.1}
>>> rdd = sc.parallelize(fractions.keys()).cartesian(sc.parallelize(range(0, 1000)))
>>> sample = dict(rdd.sampleByKey(False, fractions, 2).groupByKey().collect())
>>> 100 < len(sample["a"]) < 300 and 50 < len(sample["b"]) < 150
True
>>> max(sample["a"]) <= 999 and min(sample["a"]) >= 0
True
```

```
>>> max(sample["b"]) <= 999 and min(sample["b"]) >= 0
True
```

sampleStdev()

Compute the sample standard deviation of this RDD’s elements (which corrects for bias in estimating the standard deviation by dividing by N-1 instead of N).

```
>>> sc.parallelize([1, 2, 3]).sampleStdev()
1.0
```

sampleVariance()

Compute the sample variance of this RDD’s elements (which corrects for bias in estimating the variance by dividing by N-1 instead of N).

```
>>> sc.parallelize([1, 2, 3]).sampleVariance()
1.0
```

saveAsHadoopDataset(*conf*, *keyConverter=None*, *valueConverter=None*)

Output a Python RDD of key-value pairs (of form *RDD[(K, V)]*) to any Hadoop file system, using the old Hadoop OutputFormat API (mapred package). Keys/values are converted for output using either user specified converters or, by default, `org.apache.spark.api.python.JavaToWritableConverter`.

Parameters:

- **conf** – Hadoop job configuration, passed in as a dict
- **keyConverter** – (None by default)
- **valueConverter** – (None by default)

saveAsHadoopFile(*path*, *outputFormatClass*, *keyClass=None*, *valueClass=None*, *keyConverter=None*, *valueConverter=None*, *conf=None*, *compressionCodecClass=None*)

Output a Python RDD of key-value pairs (of form *RDD[(K, V)]*) to any Hadoop file system, using the old Hadoop OutputFormat API (mapred package). Key and value types will be inferred if not specified. Keys and values are converted for output using either user specified converters or `org.apache.spark.api.python.JavaToWritableConverter`. The `conf` is applied on top of the base Hadoop conf associated with the SparkContext of this RDD to create a merged Hadoop MapReduce job configuration for saving the data.

Parameters:

- **path** – path to Hadoop file
- **outputFormatClass** – fully qualified classname of Hadoop OutputFormat (e.g. “org.apache.hadoop.mapred.SequenceFileOutputFormat”)
- **keyClass** – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.IntWritable”, None by default)
- **valueClass** – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.Text”, None by default)
- **keyConverter** – (None by default)
- **valueConverter** – (None by default)

- **conf** – (None by default)
- **compressionCodecClass** – (None by default)

`saveAsNewAPIHadoopDataset`(*conf*, *keyConverter=None*, *valueConverter=None*)

Output a Python RDD of key-value pairs (of form *RDD[(K, V)]*) to any Hadoop file system, using the new Hadoop OutputFormat API (mapreduce package). Keys/values are converted for output using either user specified converters or, by default, `org.apache.spark.api.python.JavaToWritableConverter`.

Parameters:

- **conf** – Hadoop job configuration, passed in as a dict
- **keyConverter** – (None by default)
- **valueConverter** – (None by default)

`saveAsNewAPIHadoopFile`(*path*, *outputFormatClass*, *keyClass=None*, *valueClass=None*, *keyConverter=None*, *valueConverter=None*, *conf=None*)

Output a Python RDD of key-value pairs (of form *RDD[(K, V)]*) to any Hadoop file system, using the new Hadoop OutputFormat API (mapreduce package). Key and value types will be inferred if not specified. Keys and values are converted for output using either user specified converters or `org.apache.spark.api.python.JavaToWritableConverter`. The `conf` is applied on top of the base Hadoop conf associated with the SparkContext of this RDD to create a merged Hadoop MapReduce job configuration for saving the data.

Parameters:

- **path** – path to Hadoop file
- **outputFormatClass** – fully qualified classname of Hadoop OutputFormat (e.g. “org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat”)
- **keyClass** – fully qualified classname of key Writable class (e.g. “org.apache.hadoop.io.IntWritable”, None by default)
- **valueClass** – fully qualified classname of value Writable class (e.g. “org.apache.hadoop.io.Text”, None by default)
- **keyConverter** – (None by default)
- **valueConverter** – (None by default)
- **conf** – Hadoop job configuration, passed in as a dict (None by default)

`saveAsPickleFile`(*path*, *batchSize=10*)

Save this RDD as a SequenceFile of serialized objects. The serializer used is `pyspark.serializers.PickleSerializer`, default batch size is 10.

```
>>> tmpFile = NamedTemporaryFile(delete=True)
>>> tmpFile.close()
>>> sc.parallelize([1, 2, 'spark', 'rdd']).saveAsPickleFile(tmpFile.name, 3)
>>> sorted(sc.pickleFile(tmpFile.name, 5).map(str).collect())
['1', '2', 'rdd', 'spark']
```

`saveAsSequenceFile`(*path*, *compressionCodecClass=None*)

Output a Python RDD of key-value pairs (of form *RDD[(K, V)]*) to any Hadoop file system, using the `org.apache.hadoop.io.Writable` types that we convert from the RDD’s key and value types. The mechanism is as follows:

1. Pyrolite is used to convert pickled Python RDD into RDD of Java objects.
2. Keys and values of this Java RDD are converted to Writables and written out.

Parameters:
- **path** – path to sequence file
- **compressionCodecClass** – (None by default)

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

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

Parameters:
- **path** – path to text file
- **compressionCodecClass** – (None by default) string i.e. “org.apache.hadoop.io.compress.GzipCodec”

```
>>> tempFile = NamedTemporaryFile(delete=True)
>>> tempFile.close()
>>> sc.parallelize(range(10)).saveAsTextFile(tempFile.name)
>>> from fileinput import input
>>> from glob import glob
>>> ''.join(sorted(input(glob(tempFile.name + "/part-0000*"))))
'0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'
```

Empty lines are tolerated when saving to text files.

```
>>> tempFile2 = NamedTemporaryFile(delete=True)
>>> tempFile2.close()
>>> sc.parallelize(['', 'foo', '', 'bar', '']).saveAsTextFile(tempFile2.name)
>>> ''.join(sorted(input(glob(tempFile2.name + "/part-0000*"))))
'\n\n\nbar\nfoo\n'
```

Using compressionCodecClass

```
>>> tempFile3 = NamedTemporaryFile(delete=True)
>>> tempFile3.close()
>>> codec = "org.apache.hadoop.io.compress.GzipCodec"
>>> sc.parallelize(['foo', 'bar']).saveAsTextFile(tempFile3.name, codec)
>>> from fileinput import input, hook_compressed
>>> result = sorted(input(glob(tempFile3.name + "/part*.gz"), openhook=hook_compressed
>>> b''.join(result).decode('utf-8')
u'bar\nfoo\n'
```

setName(*name*)

Assign a name to this RDD.

```
>>> rdd1 = sc.parallelize([1, 2])
>>> rdd1.setName('RDD1').name()
u'RDD1'
```

sortBy(*keyfunc*, *ascending=True*, *numPartitions=None*)

Sorts this RDD by the given keyfunc

```
>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
>>> sc.parallelize(tmp).sortBy(lambda x: x[0]).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
>>> sc.parallelize(tmp).sortBy(lambda x: x[1]).collect()
[('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
```

sortByKey(*ascending=True*, *numPartitions=None*, *keyfunc=<function <lambda> at 0x7fc35dbcf848>*)

Sorts this RDD, which is assumed to consist of (key, value) pairs. # noqa

```
>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
>>> sc.parallelize(tmp).sortByKey().first()
('1', 3)
>>> sc.parallelize(tmp).sortByKey(True, 1).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
>>> sc.parallelize(tmp).sortByKey(True, 2).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
>>> tmp2 = [('Mary', 1), ('had', 2), ('a', 3), ('little', 4), ('lamb', 5)]
>>> tmp2.extend([('whose', 6), ('fleece', 7), ('was', 8), ('white', 9)])
>>> sc.parallelize(tmp2).sortByKey(True, 3, keyfunc=lambda k: k.lower()).collect()
[('a', 3), ('fleece', 7), ('had', 2), ('lamb', 5),...('white', 9), ('whose', 6)]
```

stats()

Return a `StatCounter` object that captures the mean, variance and count of the RDD’s elements in one operation.

stdev()

Compute the standard deviation of this RDD’s elements.

```
>>> sc.parallelize([1, 2, 3]).stdev()
0.816...
```

subtract(*other*, *numPartitions=None*)

Return each value in `self` that is not contained in `other`.

```
>>> x = sc.parallelize([("a", 1), ("b", 4), ("b", 5), ("a", 3)])
>>> y = sc.parallelize([("a", 3), ("c", None)])
>>> sorted(x.subtract(y).collect())
[('a', 1), ('b', 4), ('b', 5)]
```

subtractByKey(*other*, *numPartitions=None*)

Return each (key, value) pair in `self` that has no pair with matching key in `other`.

```
>>> x = sc.parallelize([("a", 1), ("b", 4), ("b", 5), ("a", 2)])
>>> y = sc.parallelize([("a", 3), ("c", None)])
>>> sorted(x.subtractByKey(y).collect())
[('b', 4), ('b', 5)]
```

sum()

Add up the elements in this RDD.

```
>>> sc.parallelize([1.0, 2.0, 3.0]).sum()
6.0
```

sumApprox(*timeout*, *confidence=0.95*)

Note: Experimental

Approximate operation to return the sum within a timeout or meet the confidence.

```
>>> rdd = sc.parallelize(range(1000), 10)
>>> r = sum(range(1000))
>>> abs(rdd.sumApprox(1000) - r) / r < 0.05
True
```

take(*num*)

Take the first num elements of the RDD.

It 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().

Note: this method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

```
>>> sc.parallelize([2, 3, 4, 5, 6]).cache().take(2)
[2, 3]
>>> sc.parallelize([2, 3, 4, 5, 6]).take(10)
[2, 3, 4, 5, 6]
>>> sc.parallelize(range(100), 100).filter(lambda x: x > 90).take(3)
[91, 92, 93]
```

takeOrdered(*num*, *key=None*)

Get the N elements from an RDD ordered in ascending order or as specified by the optional key function.

Note: this method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

```
>>> sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7]).takeOrdered(6)
[1, 2, 3, 4, 5, 6]
>>> sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7], 2).takeOrdered(6, key=lambda x: -x)
[10, 9, 7, 6, 5, 4]
```

takeSample(*withReplacement*, *num*, *seed=None*)

Return a fixed-size sampled subset of this RDD.

Note: This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

```
>>> rdd = sc.parallelize(range(0, 10))
>>> len(rdd.takeSample(True, 20, 1))
20
>>> len(rdd.takeSample(False, 5, 2))
5
>>> len(rdd.takeSample(False, 15, 3))
10
```

toDebugString()

A description of this RDD and its recursive dependencies for debugging.

toLocalIterator()

Return an iterator that contains all of the elements in this RDD. The iterator will consume as much memory as the largest partition in this RDD.

```
>>> rdd = sc.parallelize(range(10))
>>> [x for x in rdd.toLocalIterator()]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

top(*num*, *key=None*)

Get the top N elements from an RDD.

Note: This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

Note: It returns the list sorted in descending order.

```
>>> sc.parallelize([10, 4, 2, 12, 3]).top(1)
[12]
>>> sc.parallelize([2, 3, 4, 5, 6], 2).top(2)
[6, 5]
>>> sc.parallelize([10, 4, 2, 12, 3]).top(3, key=str)
[4, 3, 2]
```

treeAggregate(*zeroValue*, *seqOp*, *combOp*, *depth=2*)

Aggregates the elements of this RDD in a multi-level tree pattern.

Parameters: **depth** – suggested depth of the tree (default: 2)

```
>>> add = lambda x, y: x + y
>>> rdd = sc.parallelize([-5, -4, -3, -2, -1, 1, 2, 3, 4], 10)
>>> rdd.treeAggregate(0, add, add)
-5
>>> rdd.treeAggregate(0, add, add, 1)
-5
>>> rdd.treeAggregate(0, add, add, 2)
-5
>>> rdd.treeAggregate(0, add, add, 5)
-5
>>> rdd.treeAggregate(0, add, add, 10)
-5
```

treeReduce(*f*, *depth=2*)

Reduces the elements of this RDD in a multi-level tree pattern.

Parameters: **depth** – suggested depth of the tree (default: 2)

```
>>> add = lambda x, y: x + y
>>> rdd = sc.parallelize([-5, -4, -3, -2, -1, 1, 2, 3, 4], 10)
>>> rdd.treeReduce(add)
-5
>>> rdd.treeReduce(add, 1)
-5
>>> rdd.treeReduce(add, 2)
-5
>>> rdd.treeReduce(add, 5)
-5
>>> rdd.treeReduce(add, 10)
-5
```

union(*other*)

Return the union of this RDD and another one.

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

unpersist()

Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.

values()

Return an RDD with the values of each tuple.

```
>>> m = sc.parallelize([(1, 2), (3, 4)]).values()
>>> m.collect()
[2, 4]
```

variance()

Compute the variance of this RDD’s elements.

```
>>> sc.parallelize([1, 2, 3]).variance()
0.666...
```

zip(*other*)

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 (e.g. one was made through a map on the other).

```
>>> x = sc.parallelize(range(0,5))
>>> y = sc.parallelize(range(1000, 1005))
>>> x.zip(y).collect()
[(0, 1000), (1, 1001), (2, 1002), (3, 1003), (4, 1004)]
```

zipWithIndex()

Zips this RDD with its element indices.

The ordering is first based on the partition index and then the ordering of items within each partition. So the first item in the first partition gets index 0, and the last item in the last partition receives the largest index.

This method needs to trigger a spark job when this RDD contains more than one partitions.

```
>>> sc.parallelize(["a", "b", "c", "d"], 3).zipWithIndex().collect()
[('a', 0), ('b', 1), ('c', 2), ('d', 3)]
```

zipWithUniqueId()

Zips this RDD with generated unique Long ids.

Items in the kth partition will get ids k, n+k, 2*n+k, ..., where n is the number of partitions. So there may exist gaps, but this method won't trigger a spark job, which is different from `zipWithIndex`

```
>>> sc.parallelize(["a", "b", "c", "d", "e"], 3).zipWithUniqueId().collect()
[('a', 0), ('b', 1), ('c', 4), ('d', 2), ('e', 5)]
```

class pyspark.**StorageLevel**(*useDisk*, *useMemory*, *useOffHeap*, *deserialized*, *replication=1*)

Flags for controlling the storage of an RDD. Each StorageLevel records whether to use memory, whether to drop the RDD to disk if it falls out of memory, whether to keep the data in memory in a JAVA-specific serialized format, and whether to replicate the RDD partitions on multiple nodes. Also contains static constants for some commonly used storage levels, MEMORY_ONLY. Since the data is always serialized on the Python side, all the constants use the serialized formats.

DISK_ONLY = *StorageLevel(True, False, False, False, 1)*

DISK_ONLY_2 = *StorageLevel(True, False, False, False, 2)*

MEMORY_AND_DISK = *StorageLevel(True, True, False, False, 1)*

MEMORY_AND_DISK_2 = *StorageLevel(True, True, False, False, 2)*

MEMORY_AND_DISK_SER = *StorageLevel(True, True, False, False, 1)*

MEMORY_AND_DISK_SER_2 = *StorageLevel(True, True, False, False, 2)*

MEMORY_ONLY = *StorageLevel(False, True, False, False, 1)*

MEMORY_ONLY_2 = *StorageLevel(False, True, False, False, 2)*

MEMORY_ONLY_SER = *StorageLevel(False, True, False, False, 1)*

MEMORY_ONLY_SER_2 = *StorageLevel(False, True, False, False, 2)*

OFF_HEAP = *StorageLevel(True, True, True, False, 1)*

class pyspark.**Broadcast**(*sc=None*, *value=None*, *pickle_registry=None*, *path=None*)

A broadcast variable created with **SparkContext.broadcast()**. Access its value through **value**.

Examples:

```
>>> from pyspark.context import SparkContext
>>> sc = SparkContext('local', 'test')
>>> b = sc.broadcast([1, 2, 3, 4, 5])
>>> b.value
[1, 2, 3, 4, 5]
>>> sc.parallelize([0, 0]).flatMap(lambda x: b.value).collect()
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
>>> b.unpersist()
```

```
>>> large_broadcast = sc.broadcast(range(10000))
```

destroy()

Destroy all data and metadata related to this broadcast variable. Use this with caution; once a broadcast variable has been destroyed, it cannot be used again. This method blocks until destroy has completed.

dump(*value*, *f*)

load(*path*)

unpersist(*blocking=False*)

Delete cached copies of this broadcast on the executors. If the broadcast is used after this is called, it will need to be re-sent to each executor.

Parameters: **blocking** – Whether to block until unpersisting has completed

value

Return the broadcasted value

class `pyspark.`**`Accumulator`**(*aid*, *value*, *accum_param*)

A shared variable that can be accumulated, i.e., has a commutative and associative "add" operation. Worker tasks on a Spark cluster can add values to an Accumulator with the += operator, but only the driver program is allowed to access its value, using **`value`**. Updates from the workers get propagated automatically to the driver program.

While **`SparkContext`** supports accumulators for primitive data types like **`int`** and **`float`**, users can also define accumulators for custom types by providing a custom **`AccumulatorParam`** object. Refer to the doctest of this module for an example.

`add`(*term*)

Adds a term to this accumulator's value

`value`

Get the accumulator's value; only usable in driver program

class `pyspark.`**`AccumulatorParam`**

Helper object that defines how to accumulate values of a given type.

`addInPlace`(*value1*, *value2*)

Add two values of the accumulator's data type, returning a new value; for efficiency, can also update **`value1`** in place and return it.

`zero`(*value*)

Provide a "zero value" for the type, compatible in dimensions with the provided **`value`** (e.g., a zero vector)

class `pyspark.`**`MarshalSerializer`**

Serializes objects using Python's Marshal serializer:

> http://docs.python.org/2/library/marshal.html

This serializer is faster than PickleSerializer but supports fewer datatypes.

`dumps`(*obj*)

`loads`(*obj*)

class `pyspark.`**`PickleSerializer`**

Serializes objects using Python's pickle serializer:

> http://docs.python.org/2/library/pickle.html

This serializer supports nearly any Python object, but may not be as fast as more specialized serializers.

`dumps`(*obj*)

loads(*obj*, *encoding=None*)

class pyspark.**StatusTracker**(*jtracker*)

Low-level status reporting APIs for monitoring job and stage progress.

These APIs intentionally provide very weak consistency semantics; consumers of these APIs should be prepared to handle empty / missing information. For example, a job's stage ids may be known but the status API may not have any information about the details of those stages, so *getStageInfo* could potentially return *None* for a valid stage id.

To limit memory usage, these APIs only provide information on recent jobs / stages. These APIs will provide information for the last *spark.ui.retainedStages* stages and *spark.ui.retainedJobs* jobs.

getActiveJobsIds()

Returns an array containing the ids of all active jobs.

getActiveStageIds()

Returns an array containing the ids of all active stages.

getJobIdsForGroup(*jobGroup=None*)

Return a list of all known jobs in a particular job group. If *jobGroup* is None, then returns all known jobs that are not associated with a job group.

The returned list may contain running, failed, and completed jobs, and may vary across invocations of this method. This method does not guarantee the order of the elements in its result.

getJobInfo(*jobId*)

Returns a `SparkJobInfo` object, or None if the job info could not be found or was garbage collected.

getStageInfo(*stageId*)

Returns a `SparkStageInfo` object, or None if the stage info could not be found or was garbage collected.

class pyspark.**SparkJobInfo**

Exposes information about Spark Jobs.

class pyspark.**SparkStageInfo**

Exposes information about Spark Stages.

class pyspark.**Profiler**(*ctx*)

Note: DeveloperApi

PySpark supports custom profilers, this is to allow for different profilers to be used as well as outputting to different formats than what is provided in the BasicProfiler.

A custom profiler has to define or inherit the following methods:

profile - will produce a system profile of some sort. stats - return the collected stats. dump - dumps the profiles to a path add - adds a profile to the existing accumulated profile

The profiler class is chosen when creating a SparkContext

```
>>> from pyspark import SparkConf, SparkContext
>>> from pyspark import BasicProfiler
>>> class MyCustomProfiler(BasicProfiler):
...     def show(self, id):
...         print("My custom profiles for RDD:%s" % id)
...
>>> conf = SparkConf().set("spark.python.profile", "true")
>>> sc = SparkContext('local', 'test', conf=conf, profiler_cls=MyCustomProfiler)
>>> sc.parallelize(range(1000)).map(lambda x: 2 * x).take(10)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> sc.parallelize(range(1000)).count()
1000
>>> sc.show_profiles()
My custom profiles for RDD:1
My custom profiles for RDD:3
>>> sc.stop()
```

dump(*id*, *path*)

Dump the profile into path, id is the RDD id

profile(*func*)

Do profiling on the function *func*

show(*id*)

Print the profile stats to stdout, id is the RDD id

stats()

Return the collected profiling stats (pstats.Stats)

class pyspark.**BasicProfiler**(*ctx*)

BasicProfiler is the default profiler, which is implemented based on cProfile and Accumulator

profile(*func*)

Runs and profiles the method to_profile passed in. A profile object is returned.

stats()