[spark_starter]

September 12, 2021

1 Working with Apache Spark using pyspark

2 ~ Install spark and pyspark

Install spark on your server

sudo apt install spark

spark can be accessed from command line using the spark-shell command (here the shell acts as a driver program)

spark-shell -c spark.driver.bindAddress=127.0.0.1

Install the pyspark package using pip

pip install pyspark

[1]:

```
pip show pyspark
```

```
Name: pyspark
Version: 3.1.2
Summary: Apache Spark Python API
Home-page: https://github.com/apache/spark/tree/master/python
Author: Spark Developers
Author-email: dev@spark.apache.org
License: http://www.apache.org/licenses/LICENSE-2.0
Location: /home/iitp/anaconda3/lib/python3.8/site-packages
Requires: py4j
Required-by:
Note: you may need to restart the kernel to use updated packages.
```

[2]:

```
import random, os, shutil
import numpy as np
import matplotlib.pyplot as plt
import pyspark
```

3 Create spark session

a spark session must be created before working with spark.

[3]:
```
# we need to start a spark session
from pyspark.sql import SparkSession
from pyspark import SparkContext

spark = SparkSession.builder.appName('Lab6').getOrCreate()
spark
```

[3]:
```
<pyspark.sql.session.SparkSession at 0x7fcedbaceaf0>
```

4 The *context* object

[4]:
```
sc = spark.sparkContext
print('sparkContext:',sc)
```

```
sparkContext: <SparkContext master=local[*] appName=Lab6>
```

5 RDD - Resilient Distributed Datasets

RDDs are **immutable** collection of datasets that work in parallel read more@ https://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds

- **Transformations** are lazy operations on RDDs - stores actions (in a DAG) rather than actual transformation
  - read more@ https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations
- **Actions** on RDDs produce results (using the RDD DAG)
  - read more@ https://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

[6]:
```
help(pyspark.RDD)
```

```
Help on class RDD in module pyspark.rdd:

class RDD(builtins.object)
 |  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.
 |
 |  Methods defined here:
 |
 |  __add__(self, other)
 |      Return the union of this RDD and another one.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([1, 1, 2, 3])
 |      >>> (rdd + rdd).collect()
 |      [1, 1, 2, 3, 1, 1, 2, 3]
 |
 |  __getnewargs__(self)
 |
 |  __init__(self, jrdd, ctx,
jrdd_deserializer=AutoBatchedSerializer(PickleSerializer()))
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  aggregate(self, 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
 |
 |      Examples
 |      --------
 |      >>> 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(self, zeroValue, seqFunc, combFunc, numPartitions=None,
partitionFunc=<function portable_hash at 0x7fcedbed8670>)
 |      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.
 |
 |  barrier(self)
 |      Marks the current stage as a barrier stage, where Spark must launch all
tasks together.
 |      In case of a task failure, instead of only restarting the failed task,
Spark will abort the
 |      entire stage and relaunch all tasks for this stage.
 |      The barrier execution mode feature is experimental and it only handles
limited scenarios.
 |      Please read the linked SPIP and design docs to understand the
limitations and future plans.
 |
 |      .. versionadded:: 2.4.0
 |
 |      Returns
 |      -------
 |      :class:`RDDBarrier`
 |          instance that provides actions within a barrier stage.
 |
 |      See Also
 |      --------
 |      pyspark.BarrierTaskContext
 |
 |      Notes
 |      -----
 |      For additional information see
 |
 |      - `SPIP: Barrier Execution Mode
<http://jira.apache.org/jira/browse/SPARK-24374>`_
 |      - `Design Doc <https://jira.apache.org/jira/browse/SPARK-24582>`_
 |
 |      This API is experimental
 |
 |  cache(self)
 |      Persist this RDD with the default storage level (`MEMORY_ONLY`).
 |
 |  cartesian(self, 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`.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([1, 2])
 |      >>> sorted(rdd.cartesian(rdd).collect())
 |      [(1, 1), (1, 2), (2, 1), (2, 2)]
 |
 |  checkpoint(self)
 |      Mark this RDD for checkpointing. It will be saved to a file inside the
 |      checkpoint directory set with :meth:`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(self, numPartitions, shuffle=False)
 |      Return a new RDD that is reduced into `numPartitions` partitions.
 |
 |      Examples
 |      --------
 |      >>> 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(self, 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`.
 |
 |      Examples
 |      --------
 |      >>> 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(self)
 |      Return a list that contains all of the elements in this RDD.
 |
 |      Notes
 |      -----
 |      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(self)
 |      Return the key-value pairs in this RDD to the master as a dictionary.
 |
 |      Notes
 |      -----
 |      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.
 |
 |      Examples
 |      --------
 |      >>> m = sc.parallelize([(1, 2), (3, 4)]).collectAsMap()
 |      >>> m[1]
 |      2
 |      >>> m[3]
 |      4
 |
 |  collectWithJobGroup(self, groupId, description, interruptOnCancel=False)
 |      When collect rdd, use this method to specify job group.
 |
 |      .. versionadded:: 3.0.0
 |      .. deprecated:: 3.1.0
 |          Use :class:`pyspark.InheritableThread` with the pinned thread mode
enabled.
 |
 |  combineByKey(self, createCombiner, mergeValue, mergeCombiners,
numPartitions=None, partitionFunc=<function portable_hash at 0x7fcedbed8670>)
 |      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 (e.g.,
merges
 |            the lists)
 |
 |      To avoid memory allocation, both mergeValue and mergeCombiners are
allowed to
 |      modify and return their first argument instead of creating a new C.
 |
 |      In addition, users can control the partitioning of the output RDD.
 |
 |      Notes
 |    -----
 |    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]).
 |
 |    Examples
 |    --------
 |    >>> x = sc.parallelize([("a", 1), ("b", 1), ("a", 2)])
 |    >>> def to_list(a):
 |    …     return [a]
 |    …
 |    >>> def append(a, b):
 |    …     a.append(b)
 |    …     return a
 |    …
 |    >>> def extend(a, b):
 |    …     a.extend(b)
 |    …     return a
 |    …
 |    >>> sorted(x.combineByKey(to_list, append, extend).collect())
 |    [('a', [1, 2]), ('b', [1])]
 |
 |  count(self)
 |    Return the number of elements in this RDD.
 |
 |    Examples
 |    --------
 |    >>> sc.parallelize([2, 3, 4]).count()
 |    3
 |
 |  countApprox(self, timeout, confidence=0.95)
 |    Approximate version of count() that returns a potentially incomplete
 |    result within a timeout, even if not all tasks have finished.
 |
 |    Examples
 |    --------
 |    >>> rdd = sc.parallelize(range(1000), 10)
 |    >>> rdd.countApprox(1000, 1.0)
 |    1000
 |
 |  countApproxDistinct(self, relativeSD=0.05)
 |    Return approximate number of distinct elements in the RDD.
 |
 |    Parameters
 |    ----------
 |    relativeSD : float, optional
 |        Relative accuracy. Smaller values create
 |        counters that require more space.
 |        It must be greater than 0.000017.
```

```
 |
 |      Notes
 |      -----
 |      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
 |      <https://doi.org/10.1145/2452376.2452456>`_.
 |
 |      Examples
 |      --------
 |      >>> 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(self)
 |      Count the number of elements for each key, and return the result to the
 |      master as a dictionary.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
 |      >>> sorted(rdd.countByKey().items())
 |      [('a', 2), ('b', 1)]
 |
 |  countByValue(self)
 |      Return the count of each unique value in this RDD as a dictionary of
 |      (value, count) pairs.
 |
 |      Examples
 |      --------
 |      >>> sorted(sc.parallelize([1, 2, 1, 2, 2], 2).countByValue().items())
 |      [(1, 2), (2, 3)]
 |
 |  distinct(self, numPartitions=None)
 |      Return a new RDD containing the distinct elements in this RDD.
 |
 |      Examples
 |      --------
 |      >>> sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())
 |      [1, 2, 3]
 |
 |  filter(self, f)
 |      Return a new RDD containing only the elements that satisfy a predicate.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([1, 2, 3, 4, 5])
 |      >>> rdd.filter(lambda x: x % 2 == 0).collect()
 |      [2, 4]
 |
 |  first(self)
 |      Return the first element in this RDD.
 |
 |      Examples
 |      --------
 |      >>> sc.parallelize([2, 3, 4]).first()
 |      2
 |      >>> sc.parallelize([]).first()
 |      Traceback (most recent call last):
 |          …
 |      ValueError: RDD is empty
 |
 |  flatMap(self, f, preservesPartitioning=False)
 |      Return a new RDD by first applying a function to all elements of this
 |      RDD, and then flattening the results.
 |
 |      Examples
 |      --------
 |      >>> 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(self, 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.
 |
 |      Examples
 |      --------
 |      >>> 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(self, 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.
 |
 |      Examples
 |      --------
 |      >>> from operator import add
 |      >>> sc.parallelize([1, 2, 3, 4, 5]).fold(0, add)
 |      15
 |
 |  foldByKey(self, zeroValue, func, numPartitions=None, partitionFunc=<function
portable_hash at 0x7fcedbed8670>)
 |      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.).
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
 |      >>> from operator import add
 |      >>> sorted(rdd.foldByKey(0, add).collect())
 |      [('a', 2), ('b', 1)]
 |
 |  foreach(self, f)
 |      Applies a function to all elements of this RDD.
 |
 |      Examples
 |      --------
 |      >>> def f(x): print(x)
 |      >>> sc.parallelize([1, 2, 3, 4, 5]).foreach(f)
 |
 |  foreachPartition(self, f)
 |      Applies a function to each partition of this RDD.
 |
 |      Examples
 |      --------
 |      >>> def f(iterator):
 |      …     for x in iterator:
 |      …          print(x)
 |      >>> sc.parallelize([1, 2, 3, 4, 5]).foreachPartition(f)
 |
 |  fullOuterJoin(self, 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.
 |
 |      Examples
 |      --------
 |      >>> 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(self)
 |      Gets the name of the file to which this RDD was checkpointed
 |
 |      Not defined if RDD is checkpointed locally.
 |
 |  getNumPartitions(self)
 |      Returns the number of partitions in RDD
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([1, 2, 3, 4], 2)
 |      >>> rdd.getNumPartitions()
 |      2
 |
 |  getResourceProfile(self)
 |      Get the :class:`pyspark.resource.ResourceProfile` specified with this
RDD or None
 |      if it wasn't specified.
 |
 |      .. versionadded:: 3.1.0
 |
 |      Returns
 |      -------
 |      :py:class:`pyspark.resource.ResourceProfile`
 |          The the user specified profile or None if none were specified
 |
 |      Notes
 |      -----
 |      This API is experimental
 |
 |  getStorageLevel(self)
 |      Get the RDD's current storage level.
 |
 |      Examples
 |      --------
 |      >>> rdd1 = sc.parallelize([1,2])
 |      >>> rdd1.getStorageLevel()
 |      StorageLevel(False, False, False, False, 1)
 |      >>> print(rdd1.getStorageLevel())
 |      Serialized 1x Replicated
 |
 |  glom(self)
 |      Return an RDD created by coalescing all elements within each partition
 |      into a list.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([1, 2, 3, 4], 2)
 |      >>> sorted(rdd.glom().collect())
 |      [[1, 2], [3, 4]]
 |
 |  groupBy(self, f, numPartitions=None, partitionFunc=<function portable_hash
at 0x7fcedbed8670>)
 |      Return an RDD of grouped items.
 |
 |      Examples
 |      --------
 |      >>> 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(self, numPartitions=None, partitionFunc=<function portable_hash
at 0x7fcedbed8670>)
 |      Group the values for each key in the RDD into a single sequence.
 |      Hash-partitions the resulting RDD with numPartitions partitions.
 |
 |      Notes
 |      -----
 |      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.
 |
 |      Examples
 |      --------
 |      >>> 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(self, other, *others)
 |      Alias for cogroup but with support for multiple RDDs.
 |
 |      Examples
 |      --------
 |      >>> 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(self, 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) insertion 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.
 |
 |      Examples
 |      --------
 |      >>> 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(self)
 |      A unique ID for this RDD (within its SparkContext).
 |
 |  intersection(self, other)
 |      Return the intersection of this RDD and another one. The output will
 |      not contain any duplicate elements, even if the input RDDs did.
 |
 |      Notes
 |      -----
 |      This method performs a shuffle internally.
 |
 |      Examples
 |      --------
 |      >>> 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(self)
 |      Return whether this RDD is checkpointed and materialized, either
reliably or locally.
 |
 |  isEmpty(self)
 |      Returns true if and only if the RDD contains no elements at all.
 |
 |      Notes
 |      -----
 |      An RDD may be empty even when it has at least 1 partition.
 |
 |      Examples
 |      --------
 |      >>> sc.parallelize([]).isEmpty()
 |      True
 |      >>> sc.parallelize([1]).isEmpty()
 |      False
 |
 |  isLocallyCheckpointed(self)
 |      Return whether this RDD is marked for local checkpointing.
 |
 |      Exposed for testing.
 |
 |  join(self, 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.
 |
 |      Examples
 |      --------
 |      >>> 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(self, f)
 |      Creates tuples of the elements in this RDD by applying `f`.
 |
 |      Examples
 |      --------
 |      >>> 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(self)
 |      Return an RDD with the keys of each tuple.
 |
 |      Examples
 |      --------
 |      >>> m = sc.parallelize([(1, 2), (3, 4)]).keys()
 |      >>> m.collect()
 |      [1, 3]
 |
 |  leftOuterJoin(self, 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.
 |
 |      Examples
 |      --------
 |      >>> x = sc.parallelize([("a", 1), ("b", 4)])
 |      >>> y = sc.parallelize([("a", 2)])
 |      >>> sorted(x.leftOuterJoin(y).collect())
 |      [('a', (1, 2)), ('b', (4, None))]
 |
 |  localCheckpoint(self)
 |      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
:meth:`SparkContext.setCheckpointDir` is not used.
 |
 |  lookup(self, 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.
 |
 |      Examples
 |      --------
 |      >>> 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(self, f, preservesPartitioning=False)
 |      Return a new RDD by applying a function to each element of this RDD.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize(["b", "a", "c"])
 |      >>> sorted(rdd.map(lambda x: (x, 1)).collect())
 |      [('a', 1), ('b', 1), ('c', 1)]
 |
 |  mapPartitions(self, f, preservesPartitioning=False)
 |      Return a new RDD by applying a function to each partition of this RDD.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([1, 2, 3, 4], 2)
 |      >>> def f(iterator): yield sum(iterator)
 |      >>> rdd.mapPartitions(f).collect()
 |      [3, 7]
 |
 |  mapPartitionsWithIndex(self, 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.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([1, 2, 3, 4], 4)
 |      >>> def f(splitIndex, iterator): yield splitIndex
 |      >>> rdd.mapPartitionsWithIndex(f).sum()
 |      6
 |
 |  mapPartitionsWithSplit(self, 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.
 |
 |      .. deprecated:: 0.9.0
 |          use :py:meth:`RDD.mapPartitionsWithIndex` instead.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([1, 2, 3, 4], 4)
 |      >>> def f(splitIndex, iterator): yield splitIndex
 |      >>> rdd.mapPartitionsWithSplit(f).sum()
```

```
 |      6
 |
 |  mapValues(self, 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.
 |
 |      Examples
 |      --------
 |      >>> x = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b",
["grapes"])])
 |      >>> def f(x): return len(x)
 |      >>> x.mapValues(f).collect()
 |      [('a', 3), ('b', 1)]
 |
 |  max(self, key=None)
 |      Find the maximum item in this RDD.
 |
 |      Parameters
 |      ----------
 |      key : function, optional
 |          A function used to generate key for comparing
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([1.0, 5.0, 43.0, 10.0])
 |      >>> rdd.max()
 |      43.0
 |      >>> rdd.max(key=str)
 |      5.0
 |
 |  mean(self)
 |      Compute the mean of this RDD's elements.
 |
 |      Examples
 |      --------
 |      >>> sc.parallelize([1, 2, 3]).mean()
 |      2.0
 |
 |  meanApprox(self, timeout, confidence=0.95)
 |      Approximate operation to return the mean within a timeout
 |      or meet the confidence.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize(range(1000), 10)
 |      >>> r = sum(range(1000)) / 1000.0
 |      >>> abs(rdd.meanApprox(1000) - r) / r < 0.05
 |      True
 |
 |  min(self, key=None)
 |      Find the minimum item in this RDD.
 |
 |      Parameters
 |      ----------
 |      key : function, optional
 |          A function used to generate key for comparing
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([2.0, 5.0, 43.0, 10.0])
 |      >>> rdd.min()
 |      2.0
 |      >>> rdd.min(key=str)
 |      10.0
 |
 |  name(self)
 |      Return the name of this RDD.
 |
 |  partitionBy(self, numPartitions, partitionFunc=<function portable_hash at
0x7fcedbed8670>)
 |      Return a copy of the RDD partitioned using the specified partitioner.
 |
 |      Examples
 |      --------
 |      >>> 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(self, 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`).
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize(["b", "a", "c"])
 |      >>> rdd.persist().is_cached
 |      True
 |
 |  pipe(self, command, env=None, checkCode=False)
 |      Return an RDD created by piping elements to a forked external process.
 |
 |      Parameters
 |      ----------
 |      command : str
 |          command to run.
 |      env : dict, optional
 |          environment variables to set.
 |      checkCode : bool, optional
 |          whether or not to check the return value of the shell command.
 |
 |      Examples
 |      --------
 |      >>> sc.parallelize(['1', '2', '', '3']).pipe('cat').collect()
 |      ['1', '2', '', '3']
 |
 |  randomSplit(self, weights, seed=None)
 |      Randomly splits this RDD with the provided weights.
 |
 |      weights : list
 |          weights for splits, will be normalized if they don't sum to 1
 |      seed : int, optional
 |          random seed
 |
 |      Returns
 |      -------
 |      list
 |          split RDDs in a list
 |
 |      Examples
 |      --------
 |      >>> 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(self, f)
 |      Reduces the elements of this RDD using the specified commutative and
 |      associative binary operator. Currently reduces partitions locally.
 |
 |      Examples
 |      --------
 |      >>> 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(self, func, numPartitions=None, partitionFunc=<function
portable_hash at 0x7fcedbed8670>)
 |      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.
 |
 |      Examples
 |      --------
 |      >>> from operator import add
 |      >>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
 |      >>> sorted(rdd.reduceByKey(add).collect())
 |      [('a', 2), ('b', 1)]
 |
 |  reduceByKeyLocally(self, 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.
 |
 |      Examples
 |      --------
 |      >>> from operator import add
 |      >>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
 |      >>> sorted(rdd.reduceByKeyLocally(add).items())
 |      [('a', 2), ('b', 1)]
 |
 |  repartition(self, 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.
 |
 |      Examples
 |      --------
 |       >>> 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(self, numPartitions=None,
partitionFunc=<function portable_hash at 0x7fcedbed8670>, ascending=True,
keyfunc=<function RDD.<lambda> at 0x7fcedbcfc160>)
 |      Repartition the RDD according to the given partitioner and, within each
resulting partition,
 |      sort records by their keys.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([(0, 5), (3, 8), (2, 6), (0, 8), (3, 8), (1,
3)])
 |      >>> rdd2 = rdd.repartitionAndSortWithinPartitions(2, lambda x: x % 2,
True)
 |      >>> rdd2.glom().collect()
 |      [[(0, 5), (0, 8), (2, 6)], [(1, 3), (3, 8), (3, 8)]]
 |
 |  rightOuterJoin(self, 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.
 |
 |      Examples
 |      --------
 |      >>> x = sc.parallelize([("a", 1), ("b", 4)])
 |      >>> y = sc.parallelize([("a", 2)])
 |      >>> sorted(y.rightOuterJoin(x).collect())
 |      [('a', (2, 1)), ('b', (None, 4))]
 |
 |  sample(self, withReplacement, fraction, seed=None)
 |      Return a sampled subset of this RDD.
 |
 |      Parameters
 |      ----------
 |      withReplacement : bool
 |          can elements be sampled multiple times (replaced when sampled out)
 |      fraction : float
 |          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 : int, optional
 |          seed for the random number generator
 |
 |      Notes
 |      -----
 |      This is not guaranteed to provide exactly the fraction specified of the
total
 |      count of the given :class:`DataFrame`.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize(range(100), 4)
 |      >>> 6 <= rdd.sample(False, 0.1, 81).count() <= 14
 |      True
 |
 |  sampleByKey(self, 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.
 |
 |      Examples
 |      --------
 |      >>> 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(self)
 |      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).
 |
 |      Examples
 |      --------
 |      >>> sc.parallelize([1, 2, 3]).sampleStdev()
 |      1.0
 |
 |  sampleVariance(self)
 |      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).
 |
 |      Examples
 |      --------
 |      >>> sc.parallelize([1, 2, 3]).sampleVariance()
 |      1.0
 |
 |  saveAsHadoopDataset(self, 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 : dict
 |          Hadoop job configuration
 |      keyConverter : str, optional
 |          fully qualified classname of key converter (None by default)
 |      valueConverter : str, optional
 |          fully qualified classname of value converter (None by default)
 |
 |  saveAsHadoopFile(self, 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 : str
 |          path to Hadoop file
 |      outputFormatClass : str
 |          fully qualified classname of Hadoop OutputFormat
 |          (e.g. "org.apache.hadoop.mapred.SequenceFileOutputFormat")
 |      keyClass : str, optional
 |          fully qualified classname of key Writable class
 |          (e.g. "org.apache.hadoop.io.IntWritable", None by default)
 |      valueClass : str, optional
 |          fully qualified classname of value Writable class
 |          (e.g. "org.apache.hadoop.io.Text", None by default)
 |      keyConverter : str, optional
 |          fully qualified classname of key converter (None by default)
 |      valueConverter : str, optional
 |          fully qualified classname of value converter (None by default)
 |      conf : dict, optional
 |          (None by default)
 |      compressionCodecClass : str
 |          fully qualified classname of the compression codec class
 |          i.e. "org.apache.hadoop.io.compress.GzipCodec" (None by default)
 |
 |  saveAsNewAPIHadoopDataset(self, 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 : dict
 |          Hadoop job configuration
 |      keyConverter : str, optional
 |          fully qualified classname of key converter (None by default)
 |      valueConverter : str, optional
 |          fully qualified classname of value converter (None by default)
 |
 |  saveAsNewAPIHadoopFile(self, 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.
 |
 |      path : str
 |          path to Hadoop file
 |      outputFormatClass : str
 |          fully qualified classname of Hadoop OutputFormat
 |          (e.g.
"org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat")
 |      keyClass : str, optional
 |          fully qualified classname of key Writable class
 |           (e.g. "org.apache.hadoop.io.IntWritable", None by default)
 |      valueClass : str, optional
 |          fully qualified classname of value Writable class
 |          (e.g. "org.apache.hadoop.io.Text", None by default)
 |      keyConverter : str, optional
 |          fully qualified classname of key converter (None by default)
 |      valueConverter : str, optional
 |          fully qualified classname of value converter (None by default)
 |      conf : dict, optional
 |          Hadoop job configuration (None by default)
 |
 |  saveAsPickleFile(self, path, batchSize=10)
 |      Save this RDD as a SequenceFile of serialized objects. The serializer
 |      used is :class:`pyspark.serializers.PickleSerializer`, default batch
size
 |      is 10.
 |
 |      Examples
 |      --------
 |      >>> from tempfile import NamedTemporaryFile
 |      >>> 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(self, 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 : str
 |          path to sequence file
 |      compressionCodecClass : str, optional
 |          fully qualified classname of the compression codec class
 |          i.e. "org.apache.hadoop.io.compress.GzipCodec" (None by default)
 |
 |  saveAsTextFile(self, path, compressionCodecClass=None)
 |      Save this RDD as a text file, using string representations of elements.
 |
 |      Parameters
 |      ----------
 |      path : str
 |          path to text file
 |      compressionCodecClass : str, optional
 |          fully qualified classname of the compression codec class
 |          i.e. "org.apache.hadoop.io.compress.GzipCodec" (None by default)
 |
 |      Examples
 |      --------
 |      >>> from tempfile import NamedTemporaryFile
 |      >>> 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.
 |
 |      >>> from tempfile import NamedTemporaryFile
 |      >>> 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
 |
 |      >>> from tempfile import NamedTemporaryFile
 |      >>> 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')
 |      'bar\nfoo\n'
 |
 |  setName(self, name)
 |      Assign a name to this RDD.
 |
 |      Examples
 |      --------
 |      >>> rdd1 = sc.parallelize([1, 2])
 |      >>> rdd1.setName('RDD1').name()
 |      'RDD1'
 |
 |  sortBy(self, keyfunc, ascending=True, numPartitions=None)
 |      Sorts this RDD by the given keyfunc
 |
 |      Examples
 |      --------
 |      >>> 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(self, ascending=True, numPartitions=None, keyfunc=<function
RDD.<lambda> at 0x7fcedbcfc280>)
 |      Sorts this RDD, which is assumed to consist of (key, value) pairs.
 |
 |      Examples
 |      --------
 |      >>> 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(self)
 |      Return a :class:`StatCounter` object that captures the mean, variance
 |      and count of the RDD's elements in one operation.
 |
 |  stdev(self)
 |      Compute the standard deviation of this RDD's elements.
 |
 |      Examples
 |      --------
 |      >>> sc.parallelize([1, 2, 3]).stdev()
 |      0.816…
 |
 |  subtract(self, other, numPartitions=None)
 |      Return each value in `self` that is not contained in `other`.
 |
 |      Examples
 |      --------
 |      >>> 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(self, other, numPartitions=None)
 |      Return each (key, value) pair in `self` that has no pair with matching
 |      key in `other`.
 |
 |      Examples
 |      --------
 |      >>> 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(self)
 |      Add up the elements in this RDD.
 |
 |      Examples
 |      --------
 |      >>> sc.parallelize([1.0, 2.0, 3.0]).sum()
 |      6.0
 |
 |  sumApprox(self, timeout, confidence=0.95)
 |      Approximate operation to return the sum within a timeout
 |      or meet the confidence.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize(range(1000), 10)
 |      >>> r = sum(range(1000))
 |      >>> abs(rdd.sumApprox(1000) - r) / r < 0.05
 |      True
 |
 |  take(self, 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().
 |
 |      Notes
 |      -----
 |      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.
 |
 |      Examples
 |      --------
 |      >>> 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(self, num, key=None)
 |      Get the N elements from an RDD ordered in ascending order or as
 |      specified by the optional key function.
 |
 |      Notes
 |      -----
 |      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.
 |
 |      Examples
 |      --------
 |      >>> 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(self, withReplacement, num, seed=None)
 |      Return a fixed-size sampled subset of this RDD.
 |
 |      Notes
 |      -----
 |      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.
 |
 |      Examples
 |      --------
 |      >>> 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
 |
 |  toDF(self, schema=None, sampleRatio=None)
 |      Converts current :class:`RDD` into a :class:`DataFrame`
 |
 |      This is a shorthand for ``spark.createDataFrame(rdd, schema,
sampleRatio)``
 |
 |      Parameters
 |      ----------
 |      schema : :class:`pyspark.sql.types.DataType`, str or list, optional
 |          a :class:`pyspark.sql.types.DataType` or a datatype string or a list
of
 |          column names, default is None.  The data type string format equals
to
 |          :class:`pyspark.sql.types.DataType.simpleString`, except that top
level struct type can
 |          omit the ``struct<>`` and atomic types use ``typeName()`` as their
format, e.g. use
 |          ``byte`` instead of ``tinyint`` for
:class:`pyspark.sql.types.ByteType`.
 |          We can also use ``int`` as a short name for
:class:`pyspark.sql.types.IntegerType`.
 |      sampleRatio : float, optional
 |          the sample ratio of rows used for inferring
 |
 |      Returns
 |      -------
 |      :class:`DataFrame`
 |
 |      Examples
 |      --------
 |      >>> rdd.toDF().collect()
 |      [Row(name='Alice', age=1)]
 |
 |  toDebugString(self)
 |      A description of this RDD and its recursive dependencies for debugging.
 |
 |  toLocalIterator(self, prefetchPartitions=False)
 |      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.
 |      With prefetch it may consume up to the memory of the 2 largest
partitions.
 |
 |      Parameters
 |      ----------
 |      prefetchPartitions : bool, optional
 |          If Spark should pre-fetch the next partition
 |          before it is needed.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize(range(10))
 |      >>> [x for x in rdd.toLocalIterator()]
 |      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
 |
 |  top(self, num, key=None)
 |      Get the top N elements from an RDD.
 |
 |      Notes
 |      -----
 |      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.
 |
 |      It returns the list sorted in descending order.
 |
 |      Examples
 |      --------
 |      >>> 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(self, zeroValue, seqOp, combOp, depth=2)
 |      Aggregates the elements of this RDD in a multi-level tree
 |      pattern.
 |
 |      depth : int, optional
 |          suggested depth of the tree (default: 2)
 |
 |      Examples
 |      --------
 |      >>> 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(self, f, depth=2)
 |      Reduces the elements of this RDD in a multi-level tree pattern.
 |
 |      Parameters
 |      ----------
 |      f : function
 |      depth : int, optional
 |          suggested depth of the tree (default: 2)
 |
 |      Examples
 |      --------
 |      >>> 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(self, other)
 |      Return the union of this RDD and another one.
 |
 |      Examples
 |      --------
 |      >>> rdd = sc.parallelize([1, 1, 2, 3])
```

```
 |      >>> rdd.union(rdd).collect()
 |      [1, 1, 2, 3, 1, 1, 2, 3]
 |
 |  unpersist(self, blocking=False)
 |      Mark the RDD as non-persistent, and remove all blocks for it from
 |      memory and disk.
 |
 |      .. versionchanged:: 3.0.0
 |         Added optional argument `blocking` to specify whether to block until
all
 |         blocks are deleted.
 |
 |  values(self)
 |      Return an RDD with the values of each tuple.
 |
 |      Examples
 |      --------
 |      >>> m = sc.parallelize([(1, 2), (3, 4)]).values()
 |      >>> m.collect()
 |      [2, 4]
 |
 |  variance(self)
 |      Compute the variance of this RDD's elements.
 |
 |      Examples
 |      --------
 |      >>> sc.parallelize([1, 2, 3]).variance()
 |      0.666…
 |
 |  withResources(self, profile)
 |      Specify a :class:`pyspark.resource.ResourceProfile` to use when
calculating this RDD.
 |      This is only supported on certain cluster managers and currently
requires dynamic
 |      allocation to be enabled. It will result in new executors with the
resources specified
 |      being acquired to calculate the RDD.
 |
 |      .. versionadded:: 3.1.0
 |
 |      Notes
 |      -----
 |      This API is experimental
 |
 |  zip(self, 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).
 |
 |      Examples
 |      --------
 |      >>> 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(self)
 |      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.
 |
 |      Examples
 |      --------
 |      >>> sc.parallelize(["a", "b", "c", "d"], 3).zipWithIndex().collect()
 |      [('a', 0), ('b', 1), ('c', 2), ('d', 3)]
 |
 |  zipWithUniqueId(self)
 |      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
 |      :meth:`zipWithIndex`.
 |
 |      Examples
 |      --------
 |      >>> sc.parallelize(["a", "b", "c", "d", "e"],
3).zipWithUniqueId().collect()
 |      [('a', 0), ('b', 1), ('c', 4), ('d', 2), ('e', 5)]
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties defined here:
 |
 |  context
 |      The :class:`SparkContext` that this RDD was created on.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
```

```
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

6 Creating RDDs

6.1 creating from sc.parallelize()

[7]:
```
RDD_array1 = sc.parallelize(np.random.randint(0,10,size=10))
```

6.2 apply transformation and actions

[8]:
```
RDD_array2 = RDD_array1.map(lambda x: x*2) # 'map' is a transformation
```

[9]:
```
a1 = RDD_array1.collect() # 'collect' is an action
a2 = RDD_array2.collect()
print( a1,  a2)
```

```
[7, 9, 2, 0, 4, 1, 2, 0, 5, 0] [14, 18, 4, 0, 8, 2, 4, 0, 10, 0]
```

[10]:
```
a3 = RDD_array2.reduce(lambda x,y: x+y)  # 'reduce' is an action
print(a3)
```

```
60
```

6.3 creating from files or external objects

[11]:
```
RDD_F = sc.textFile("Apple_stock.csv")
```

[12]:
```
cf = RDD_F.count() # 'count' is an action
print('Count:',cf)

data = RDD_F.collect() # 'collect' is an action
print(type(data), len(data))

samples = RDD_F.takeSample(True, 3)  # takeSample is an action
print('Samples\n', type(samples), len(samples), samples)

first = RDD_F.first()
```

```
print('Header:',first) #<---- shall be used for schema
```

```
Count: 1597
<class 'list'> 1597
Samples
 <class 'list'> 3 ['2015-11-16,28.559999465942383,27.75,27.844999313354492,28.54
5000076293945,152426800.0,26.322452545166016', '2011-01-03,11.795000076293945,11
.601428985595703,11.630000114440918,11.770357131958008,445138400.0,10.1062202453
61328', '2012-09-10,24.403213500976562,23.64642906188965,24.301786422729492,23.6
69286727905273,487998000.0,20.410099029541016']
Header: Date,High,Low,Open,Close,Volume,Adj Close
```

7 mapper function used to map values to data frame

[13]:

```
def mapper(xs):
    global first
    if xs!=first:
        x = xs.split(",")
        return [str(x[0]), float(x[1]), float(x[2]), float(x[3]), float(x[4]),␣
 ↪float(x[5]), float(x[6])]
    else:
        pass
```

7.1 create data frame using mapper

[14]:

```
dataset = RDD_F.map(mapper).filter(lambda x: x!=None).toDF(first.split(','))
print(type(dataset), dataset.count())
dataset.printSchema()
dataset.show() # or use .describe()
```

```
<class 'pyspark.sql.dataframe.DataFrame'> 1596
root
 |-- Date: string (nullable = true)
 |-- High: double (nullable = true)
 |-- Low: double (nullable = true)
 |-- Open: double (nullable = true)
 |-- Close: double (nullable = true)
 |-- Volume: double (nullable = true)
 |-- Adj Close: double (nullable = true)

+----------+-----------------+-----------------+-----------------+-------------
---+----------+------------------+
|      Date|             High|              Low|             Open|
Close|    Volume|         Adj Close|
+----------+-----------------+-----------------+-----------------+--------------
---+----------+------------------+
|2010-08-02|9.378213882446289|9.272143363952637|
9.30142879486084|9.351785659790039|4.280556E8| 8.029596328735352|
|2010-08-03|9.402142524719238|9.265000343322754|9.321785926818848|9.354642868041
992|4.176536E8| 8.032052993774414|
|2010-08-04|9.438570976257324|9.296786308288574|
9.3871431350708|9.392143249511719|4.203752E8| 8.064249038696289|
|2010-08-05|9.399286270141602|9.305356979370117|
9.34749984741211|9.346428871154785|2.890972E8| 8.024996757507324|
|2010-08-06|9.338929176330566|9.201070785522461|9.277856826782227|9.288928985595
703|4.448976E8| 7.975627899169922|
|2010-08-09|9.362500190734863|9.270357131958008|9.338570594787598|9.348214149475
098| 3.03128E8| 8.026529312133789|
|2010-08-10|9.301786422729492|9.198213577270508|9.280357360839844|9.264642715454
102|  4.5192E8| 7.954774856567383|
|2010-08-11|9.131786346435547|8.921786308288574|9.121429443359375|8.935357093811
035|6.200544E8| 7.672046184539795|
|2010-08-12|9.039285659790039|8.789999961853027|8.810357093811035|8.992500305175
781|5.349204E8|7.7211103439331055|
|2010-08-13| 8.99571418762207|8.896071434020996|8.987500190734863|8.896429061889
648|3.548692E8| 7.638622283935547|
|2010-08-16|8.928929328918457|8.807856559753418|8.842143058776855|
8.84428596496582|  3.1843E8|7.5938496589660645|
|2010-08-17|9.093929290771484|8.899999618530273|8.931428909301758|8.998929023742
676|4.226404E8| 7.726629257202148|
|2010-08-18|9.095356941223145|8.984999656677246|9.012857437133789|9.038213729858
398| 3.39696E8|7.7603607177734375|
|2010-08-19|9.052857398986816|8.881428718566895|9.029999732971191|8.924285888671
875| 4.26706E8| 7.662538051605225|
|2010-08-20|9.068571090698242| 8.89285659790039|
8.90678596496582|8.915714263916016|  3.8423E8| 7.655179500579834|
|2010-08-23|              9.0|8.758929252624512|8.992500305175781|8.778571128845
215|4.140416E8|7.5374250411987305|
|2010-08-24|8.678570747375488|8.523214340209961|8.666786193847656|8.568928718566
895|6.025656E8| 7.357422828674316|
|2010-08-25|8.713929176330566|8.471428871154785|8.501428604125977|8.674642562866
211|5.968676E8| 7.448192596435547|
|2010-08-26|8.776785850524902|8.581428527832031|8.766071319580078|8.581428527832
031|4.665052E8| 7.368154048919678|
|2010-08-27|8.664643287658691|8.412857055664062|8.633929252624512|
8.62928581237793|5.483912E8| 7.409246921539307|
+----------+-----------------+-----------------+-----------------+--------------
---+----------+------------------+
only showing top 20 rows
```

8 Linear Regression Example

Not using spark.mllib

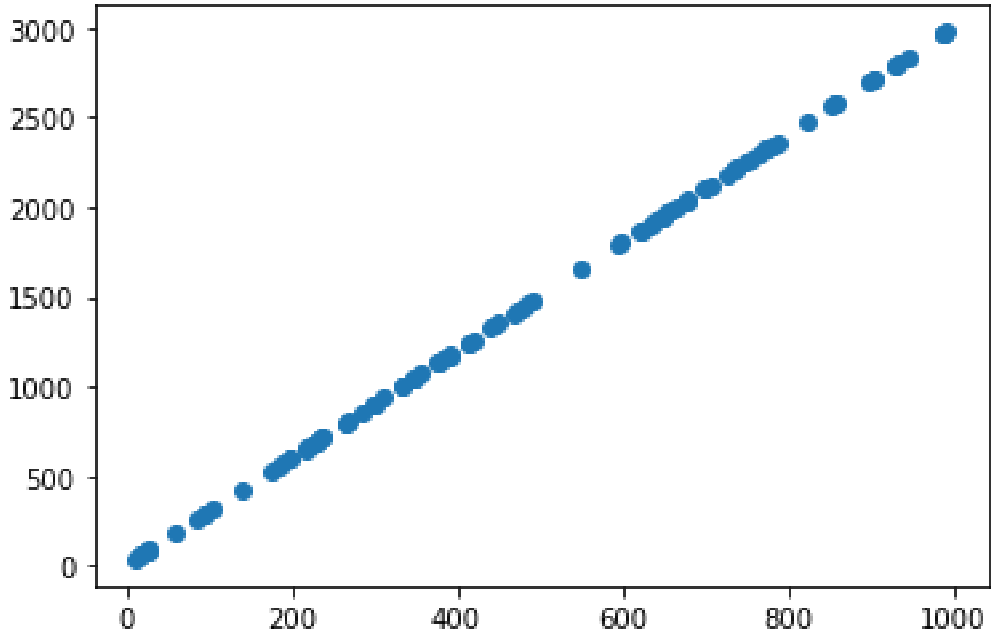
[15]:

```
# a line is y = mx + c where m and c are the params
ground_m, ground_c = random.randint(0,5), random.randint(0,5) # the ground truth

def line(x):
    global ground_m, ground_c
    return ground_m*x + ground_c

# generate at least 100 samples and save to a file
rdd_x = sc.parallelize(np.random.randint(0,1000,size=100))
rdd_y = rdd_x.map(line)

# generate input data and save to file
data_x, data_y = rdd_x.collect(), rdd_y.collect() #<-- 'collect' is an action
plt.scatter(data_x, data_y)
plt.show()
```



8.1 Estimate using keras

[16]:

```
from tensorflow.keras import Model  # always use this approach instead of␣
 ↪Sequential
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam, SGD, RMSprop

# create a single layer model
input_layer = Input((1,))
output_layer = Dense(1)(input_layer)
model = Model(inputs=input_layer, outputs=output_layer)
model.compile(loss='mse', optimizer=Adam(learning_rate=0.1), metrics=[])
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 1)]               0
_________________________________________________________________
dense (Dense)                (None, 1)                 2
=================================================================
Total params: 2
Trainable params: 2
Non-trainable params: 0
_________________________________________________________________
```

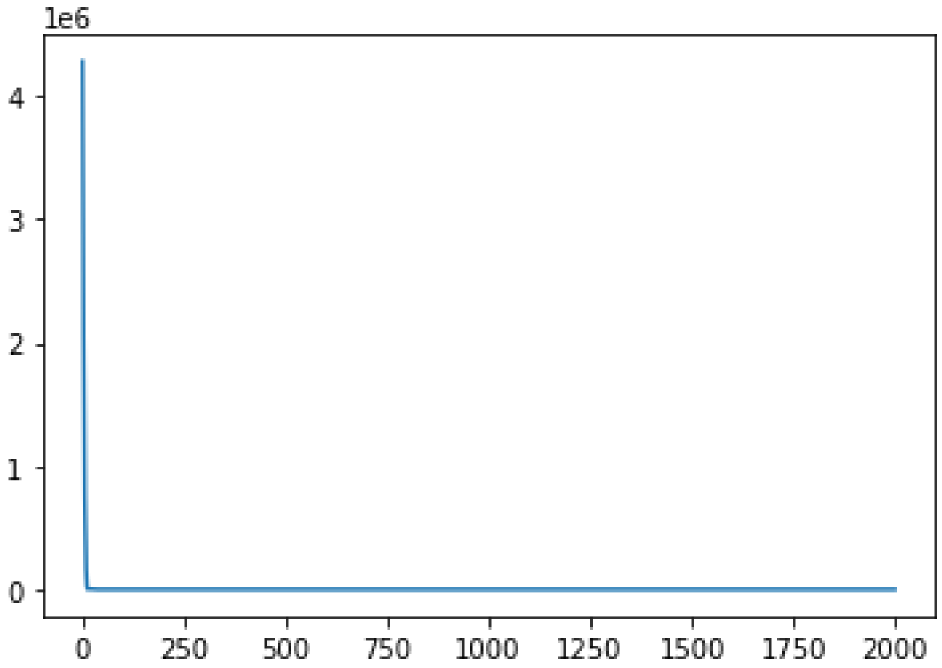
9 Train model

[17]:

```
lossv=[]
for ep in range(100):
    #print(ep+1)
    hist=model.fit(x=np.array(rdd_x.collect()), #<--- new data every time␣
 ↪collect() is called
                   y=np.array(rdd_y.collect()),
                   batch_size=32, epochs=20,verbose=0)
    lossv.extend(hist.history['loss'])
plt.plot(lossv)
```

[17]:

```
[<matplotlib.lines.Line2D at 0x7fce94424c40>]
```



10 Check for convergence

check if estimated parameters are close enough to ground truth

```
[18]: model_w8 = model.get_weights()
      est_m, est_c = model_w8[0][0],model_w8[1][0]
      print('Model Estimate:', est_m, est_c)
      print('Ground Truth:','['+str(ground_m)+']', ground_c)

      print('Delta:',ground_m-est_m, ground_c-est_c)
```

```
Model Estimate: [3.] 3.9999743
Ground Truth: [3] 4
Delta: [0.] 2.574920654296875e-05
```

11 Using spark DataFrames in general

12 Reading Data

[19]:

```
dataset = spark.read.option('header',
                            'true' #<---- the csv has a header
                           ).csv("Apple_stock.csv",
                                 inferSchema=True) # add infer schema to load in␣
 ↪proper data type

print(type(dataset), dataset.count())
dataset.printSchema()
dataset.show() # or use .describe()
```

```
<class 'pyspark.sql.dataframe.DataFrame'> 1596
root
 |-- Date: string (nullable = true)
 |-- High: double (nullable = true)
 |-- Low: double (nullable = true)
 |-- Open: double (nullable = true)
 |-- Close: double (nullable = true)
 |-- Volume: double (nullable = true)
 |-- Adj Close: double (nullable = true)

+----------+-----------------+-----------------+-----------------+--------------
---+----------+------------------+
|      Date|             High|              Low|             Open|
Close|    Volume|         Adj Close|
+----------+-----------------+-----------------+-----------------+--------------
---+----------+------------------+
|2010-08-02|9.378213882446289|9.272143363952637|
9.30142879486084|9.351785659790039|4.280556E8| 8.029596328735352|
|2010-08-03|9.402142524719238|9.265000343322754|9.321785926818848|9.354642868041
992|4.176536E8| 8.032052993774414|
|2010-08-04|9.438570976257324|9.296786308288574|
9.3871431350708|9.392143249511719|4.203752E8| 8.064249038696289|
|2010-08-05|9.399286270141602|9.305356979370117|
9.34749984741211|9.346428871154785|2.890972E8| 8.024996757507324|
|2010-08-06|9.338929176330566|9.201070785522461|9.277856826782227|9.288928985595
703|4.448976E8| 7.975627899169922|
|2010-08-09|9.362500190734863|9.270357131958008|9.338570594787598|9.348214149475
098| 3.03128E8| 8.026529312133789|
|2010-08-10|9.301786422729492|9.198213577270508|9.280357360839844|9.264642715454
102|  4.5192E8| 7.954774856567383|
|2010-08-11|9.131786346435547|8.921786308288574|9.121429443359375|8.935357093811
035|6.200544E8| 7.672046184539795|
|2010-08-12|9.039285659790039|8.789999961853027|8.810357093811035|8.992500305175
781|5.349204E8|7.7211103439331055|
|2010-08-13| 8.99571418762207|8.896071434020996|8.987500190734863|8.896429061889
648|3.548692E8| 7.638622283935547|
|2010-08-16|8.928929328918457|8.807856559753418|8.842143058776855|
8.84428596496582|  3.1843E8|7.5938496589660645|
|2010-08-17|9.093929290771484|8.899999618530273|8.931428909301758|8.998929023742
676|4.226404E8| 7.726629257202148|
|2010-08-18|9.095356941223145|8.984999656677246|9.012857437133789|9.038213729858
398| 3.39696E8|7.7603607177734375|
|2010-08-19|9.052857398986816|8.881428718566895|9.029999732971191|8.924285888671
875| 4.26706E8| 7.662538051605225|
|2010-08-20|9.068571090698242| 8.89285659790039|
8.90678596496582|8.915714263916016|  3.8423E8| 7.655179500579834|
|2010-08-23|              9.0|8.758929252624512|8.992500305175781|8.778571128845
215|4.140416E8|7.5374250411987305|
|2010-08-24|8.678570747375488|8.523214340209961|8.666786193847656|8.568928718566
895|6.025656E8| 7.357422828674316|
|2010-08-25|8.713929176330566|8.471428871154785|8.501428604125977|8.674642562866
211|5.968676E8| 7.448192596435547|
|2010-08-26|8.776785850524902|8.581428527832031|8.766071319580078|8.581428527832
031|4.665052E8| 7.368154048919678|
|2010-08-27|8.664643287658691|8.412857055664062|8.633929252624512|
8.62928581237793|5.483912E8| 7.409246921539307|
+----------+-----------------+-----------------+-----------------+--------------
---+----------+------------------+
only showing top 20 rows
```

[20]:

```
dataset.select(['Date','Volume']).show()
```

```
+----------+----------+
|      Date|    Volume|
+----------+----------+
|2010-08-02|4.280556E8|
|2010-08-03|4.176536E8|
|2010-08-04|4.203752E8|
|2010-08-05|2.890972E8|
|2010-08-06|4.448976E8|
|2010-08-09| 3.03128E8|
|2010-08-10|  4.5192E8|
|2010-08-11|6.200544E8|
|2010-08-12|5.349204E8|
|2010-08-13|3.548692E8|
|2010-08-16|  3.1843E8|
|2010-08-17|4.226404E8|
|2010-08-18| 3.39696E8|
|2010-08-19| 4.26706E8|
|2010-08-20|  3.8423E8|
|2010-08-23|4.140416E8|
|2010-08-24|6.025656E8|
|2010-08-25|5.968676E8|
|2010-08-26|4.665052E8|
|2010-08-27|5.483912E8|
+----------+----------+
only showing top 20 rows
```

13 Filter Operation

[21]:

```
dataset.filter("Volume>=400000000").select(["Date","Volume","Adj Close"]).show()
```

```
+----------+----------+------------------+
|      Date|    Volume|         Adj Close|
+----------+----------+------------------+
|2010-08-02|4.280556E8| 8.029596328735352|
|2010-08-03|4.176536E8| 8.032052993774414|
|2010-08-04|4.203752E8| 8.064249038696289|
|2010-08-06|4.448976E8| 7.975627899169922|
|2010-08-10|  4.5192E8| 7.954774856567383|
|2010-08-11|6.200544E8| 7.672046184539795|
|2010-08-12|5.349204E8|7.7211103439331055|
|2010-08-17|4.226404E8| 7.726629257202148|
|2010-08-19| 4.26706E8| 7.662538051605225|
|2010-08-23|4.140416E8|7.5374250411987305|
|2010-08-24|6.025656E8| 7.357422828674316|
|2010-08-25|5.968676E8| 7.448192596435547|
|2010-08-26|4.665052E8| 7.368154048919678|
|2010-08-27|5.483912E8| 7.409246921539307|
|2010-08-31|4.207868E8|7.4546308517456055|
|2010-09-01|6.970376E8|7.6763386726379395|
|2010-09-02|4.154276E8| 7.732760429382324|
|2010-09-03|5.207888E8| 7.935146808624268|
|2010-09-08|5.265512E8| 8.062408447265625|
|2010-09-09|4.385752E8| 8.067007064819336|
+----------+----------+------------------+
only showing top 20 rows
```