

 [learn-co-curriculum](#) / [dsc-resilient-distributed-datasets-rdd-lab](#) Public [View license](#) 1 star  152 forks Star Watch ▾[Code](#) [Issues 1](#) [Pull requests](#) [Actions](#) [Projects](#) [Security](#) [Insights](#) solution ▾

...

This branch is [13 commits ahead](#), [15 commits behind](#) master.

hoffm386 fix H1 ...

on Aug 4 17

[View code](#) README.md

# Resilient Distributed Datasets (RDDs) - Lab

Resilient Distributed Datasets (RDD) are fundamental data structures of Spark. An RDD is essentially the Spark representation of a set of data, spread across multiple machines, with APIs to let you act on it. An RDD can come from any data source, e.g. text files, a database, a JSON file, etc.

## Objectives

You will be able to:

- Apply the map(func) transformation to a given function on all elements of an RDD in different partitions
- Apply a map transformation for all elements of an RDD
- Compare the difference between a transformation and an action within RDDs
- Use collect(), count(), and take() actions to trigger spark transformations
- Use filter to select data that meets certain specifications within an RDD

- Set number of partitions for parallelizing RDDs
- Create RDDs from Python collections

## What are RDDs?

---

To get a better understanding of RDDs, let's break down each one of the components of the acronym RDD:

**Resilient:** RDDs are considered "resilient" because they have built-in fault tolerance. This means that even if one of the nodes goes offline, RDDs will be able to restore the data. This is already a huge advantage compared to standard storage. If a standard computer dies while performing an operation, all of its memory will be lost in the process. With RDDs, multiple nodes can go offline, and the action will still be held in working memory.

**Distributed:** The data is contained on multiple nodes of a cluster-computing operation. It is efficiently partitioned to allow for parallelism.

**Dataset:** The dataset has been \* partitioned \* across the multiple nodes.

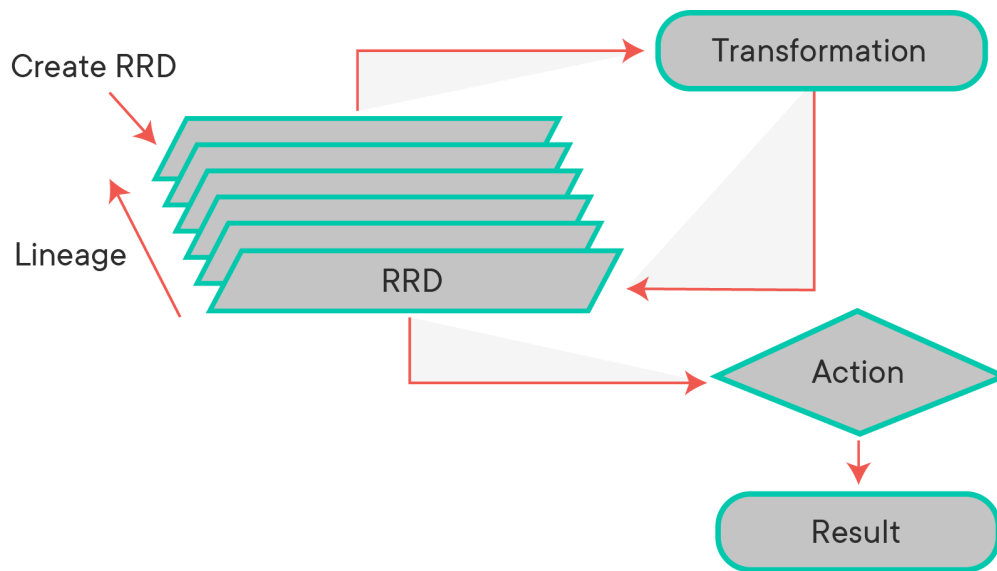
RDDs are the building block upon which more high-level Spark operations are based upon. Chances are, if you are performing an action using Spark, the operation involves RDDs.

Key Characteristics of RDDs:

- **Immutable:** Once an RDD is created, it cannot be modified.
- **Lazily Evaluated:** RDDs will not be evaluated until an action is triggered. Essentially, when RDDs are created, they are programmed to perform some action, but that function will not get activated until it is explicitly called. The reason for lazy evaluation is that allows users to organize the actions of their Spark program into smaller actions. It also saves unnecessary computation and memory load.
- **In-Memory:** The operations in Spark are performed in-memory rather than in the database. This is what allows Spark to perform fast operations with very large quantities of data.

## RDD Transformations vs Actions

In Spark, we first create a **base RDD** and then apply one or more transformations to that base RDD following our processing needs. Being immutable means, **once an RDD is created, it cannot be changed**. As a result, **each transformation of an RDD creates a new RDD**. Finally, we can apply one or more **actions** to the RDDs. Spark uses lazy evaluation, so transformations are not actually executed until an action occurs.



## Transformations

Transformations create a new dataset from an existing one by passing each dataset element through a function and returning a new RDD representing the results. In short, creating an RDD from an existing RDD is 'transformation'. All transformations in Spark are lazy. They do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result that needs to be returned to the driver program. A transformation is an RDD that returns another RDD, like `map`, `flatMap`, `filter`, `reduceByKey`, `join`, `cogroup`, etc.

## Actions

Actions return final results of RDD computations. Actions trigger execution using lineage graph to load the data into original RDD and carry out all intermediate transformations and return the final results to the driver program or writes it out to the file system. An action returns a value (to a Spark driver - the user program).

Here are some key transformations and actions that we will explore.

Transformations	Actions
<code>map(func)</code>	<code>reduce(func)</code>
<code>filter(func)</code>	<code>collect()</code>
<code>groupByKey()</code>	<code>count()</code>
<code>reduceByKey(func)</code>	<code>first()</code>

Transformations	Actions
mapValues(func)	take()
sample()	countByKey()
distinct()	foreach(func)
sortByKey()	

Let's see how transformations and actions work through a simple example. In this example, we will perform several actions and transformations on RDDs in order to obtain a better understanding of Spark processing.

## Create a Python collection

We need some data to start experimenting with RDDs. Let's create some sample data and see how RDDs handle it. To practice working with RDDs, we're going to use a simple Python list.

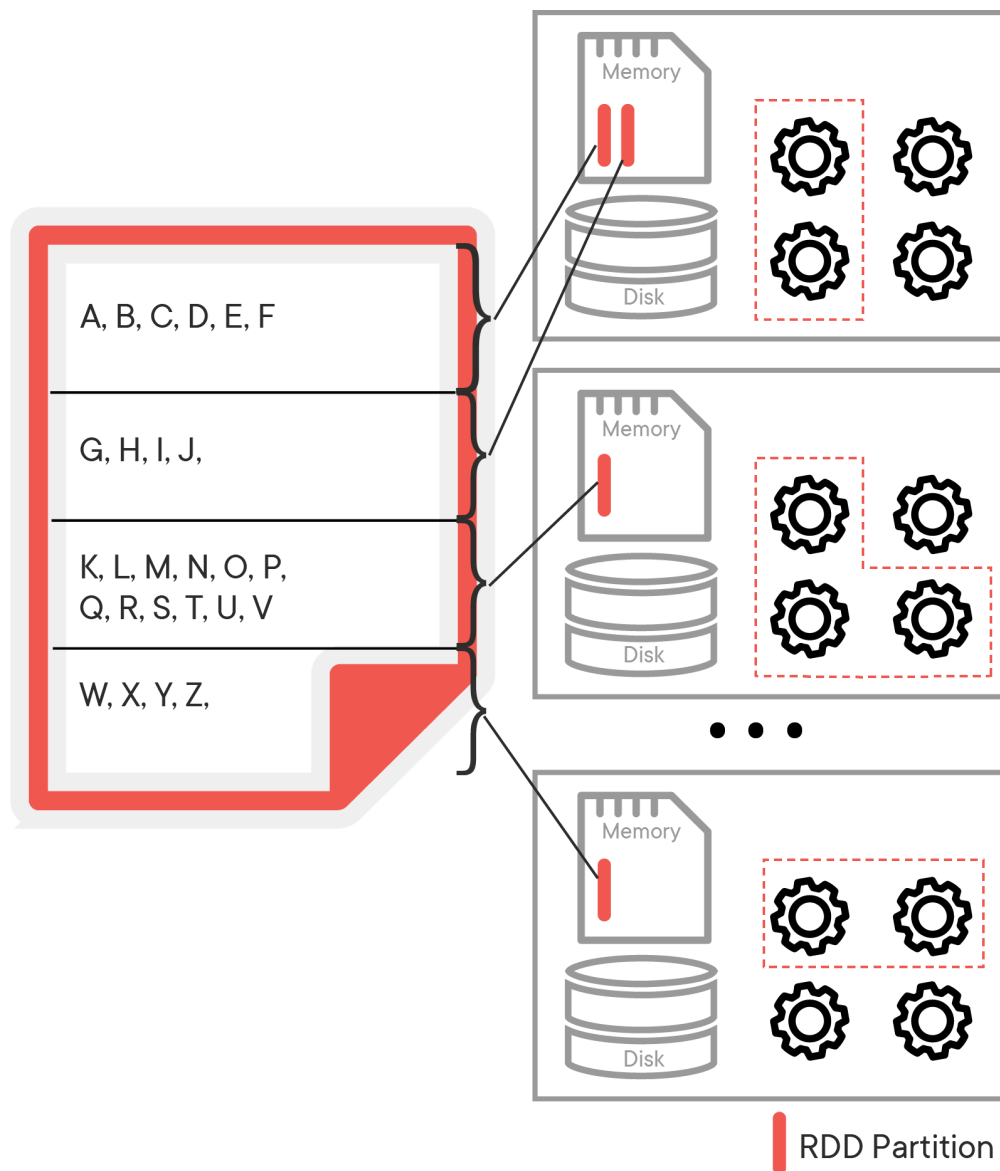
- Create a Python list `data` of integers between 1 and 1000 using the `range()` function.
- Sanity check: confirm the length of the list (it should be 1000)

```
nums = list(range(1,1001))  
len(nums)
```

```
1000
```

## Initialize an RDD

When using Spark to make computations, datasets are treated as lists of entries. Those lists are split into different partitions across different cores or different computers. Each list of data held in memory is a partition of the RDD. The reason why Spark is able to make computations far faster than other big data processing languages is that it allows all data to be stored **in-memory**, which allows for easy access to the data and, in turn, high-speed processing. Here is an example of how the alphabet might be split into different RDDs and held across a distributed collection of nodes:



To initialize an RDD, first import `pyspark` and then create a `SparkContext` assigned to the variable `sc`. Use `'local[*]'` as the master.

```
import pyspark
sc = pyspark.SparkContext('local[*]', 'RDD practice')
```

Once you've created the `SparkContext`, you can use the `.parallelize()` method to create an RDD that will distribute the list of numbers across multiple cores. Here, create one called `rdd` with 10 partitions using `data` as the collection you are parallelizing.

```
rdd = sc.parallelize(nums, numSlices=10)
print(type(rdd))
```

```
<class 'pyspark.rdd.RDD'>
```

Determine how many partitions are being used with this RDD with the `.getNumPartitions()` method.

```
rdd.getNumPartitions()
```

```
10
```

## Basic descriptive RDD actions

Let's perform some basic operations on our RDD. In the cell below, use the methods:

- `count` : returns the total count of items in the RDD
- `first` : returns the first item in the RDD
- `take` : returns the first `n` items in the RDD
- `top` : returns the top `n` items
- `collect` : returns everything from your RDD

It's important to note that in a big data context, calling the `collect` method will often take a very long time to execute and should be handled with care!

```
rdd.count()
```

```
1000
```

```
rdd.first()
```

```
1
```

```
rdd.take(10)
```

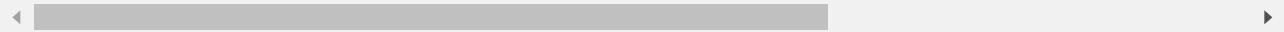
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```



```
rdd.top(10)
```

```
[1000, 999, 998, 997, 996, 995, 994, 993, 992, 991]
```

```
## Note: When you are dealing with big data, this could make your computer crash! It  
rdd.collect()
```



```
[1,  
 2,  
 3,  
 4,  
 5,  
 6,  
 7,  
 8,  
 9,  
10,  
11,  
12,  
13,  
14,  
15,  
16,  
17,  
18,  
19,  
20,  
21,  
22,  
23,  
24,  
25,  
26,  
27,  
28,  
29,  
30,  
31,  
32,  
33,  
34,  
35,  
36,
```

37,  
38,  
39,  
40,  
41,  
42,  
43,  
44,  
45,  
46,  
47,  
48,  
49,  
50,  
51,  
52,  
53,  
54,  
55,  
56,  
57,  
58,  
59,  
60,  
61,  
62,  
63,  
64,  
65,  
66,  
67,  
68,  
69,  
70,  
71,  
72,  
73,  
74,  
75,  
76,  
77,  
78,  
79,  
80,  
81,  
82,  
83,  
84,  
85,



86,  
87,  
88,  
89,  
90,  
91,  
92,  
93,  
94,  
95,  
96,  
97,  
98,  
99,  
100,  
101,  
102,  
103,  
104,  
105,  
106,  
107,  
108,  
109,  
110,  
111,  
112,  
113,  
114,  
115,  
116,  
117,  
118,  
119,  
120,  
121,  
122,  
123,  
124,  
125,  
126,  
127,  
128,  
129,  
130,  
131,  
132,  
133,  
134,

135,  
136,  
137,  
138,  
139,  
140,  
141,  
142,  
143,  
144,  
145,  
146,  
147,  
148,  
149,  
150,  
151,  
152,  
153,  
154,  
155,  
156,  
157,  
158,  
159,  
160,  
161,  
162,  
163,  
164,  
165,  
166,  
167,  
168,  
169,  
170,  
171,  
172,  
173,  
174,  
175,  
176,  
177,  
178,  
179,  
180,  
181,  
182,  
183,

184,  
185,  
186,  
187,  
188,  
189,  
190,  
191,  
192,  
193,  
194,  
195,  
196,  
197,  
198,  
199,  
200,  
201,  
202,  
203,  
204,  
205,  
206,  
207,  
208,  
209,  
210,  
211,  
212,  
213,  
214,  
215,  
216,  
217,  
218,  
219,  
220,  
221,  
222,  
223,  
224,  
225,  
226,  
227,  
228,  
229,  
230,  
231,  
232,

233,  
234,  
235,  
236,  
237,  
238,  
239,  
240,  
241,  
242,  
243,  
244,  
245,  
246,  
247,  
248,  
249,  
250,  
251,  
252,  
253,  
254,  
255,  
256,  
257,  
258,  
259,  
260,  
261,  
262,  
263,  
264,  
265,  
266,  
267,  
268,  
269,  
270,  
271,  
272,  
273,  
274,  
275,  
276,  
277,  
278,  
279,  
280,  
281,

282,  
283,  
284,  
285,  
286,  
287,  
288,  
289,  
290,  
291,  
292,  
293,  
294,  
295,  
296,  
297,  
298,  
299,  
300,  
301,  
302,  
303,  
304,  
305,  
306,  
307,  
308,  
309,  
310,  
311,  
312,  
313,  
314,  
315,  
316,  
317,  
318,  
319,  
320,  
321,  
322,  
323,  
324,  
325,  
326,  
327,  
328,  
329,  
330,

331,  
332,  
333,  
334,  
335,  
336,  
337,  
338,  
339,  
340,  
341,  
342,  
343,  
344,  
345,  
346,  
347,  
348,  
349,  
350,  
351,  
352,  
353,  
354,  
355,  
356,  
357,  
358,  
359,  
360,  
361,  
362,  
363,  
364,  
365,  
366,  
367,  
368,  
369,  
370,  
371,  
372,  
373,  
374,  
375,  
376,  
377,  
378,  
379,

380,  
381,  
382,  
383,  
384,  
385,  
386,  
387,  
388,  
389,  
390,  
391,  
392,  
393,  
394,  
395,  
396,  
397,  
398,  
399,  
400,  
401,  
402,  
403,  
404,  
405,  
406,  
407,  
408,  
409,  
410,  
411,  
412,  
413,  
414,  
415,  
416,  
417,  
418,  
419,  
420,  
421,  
422,  
423,  
424,  
425,  
426,  
427,  
428,

429,  
430,  
431,  
432,  
433,  
434,  
435,  
436,  
437,  
438,  
439,  
440,  
441,  
442,  
443,  
444,  
445,  
446,  
447,  
448,  
449,  
450,  
451,  
452,  
453,  
454,  
455,  
456,  
457,  
458,  
459,  
460,  
461,  
462,  
463,  
464,  
465,  
466,  
467,  
468,  
469,  
470,  
471,  
472,  
473,  
474,  
475,  
476,  
477,



478,  
479,  
480,  
481,  
482,  
483,  
484,  
485,  
486,  
487,  
488,  
489,  
490,  
491,  
492,  
493,  
494,  
495,  
496,  
497,  
498,  
499,  
500,  
501,  
502,  
503,  
504,  
505,  
506,  
507,  
508,  
509,  
510,  
511,  
512,  
513,  
514,  
515,  
516,  
517,  
518,  
519,  
520,  
521,  
522,  
523,  
524,  
525,  
526,

527,  
528,  
529,  
530,  
531,  
532,  
533,  
534,  
535,  
536,  
537,  
538,  
539,  
540,  
541,  
542,  
543,  
544,  
545,  
546,  
547,  
548,  
549,  
550,  
551,  
552,  
553,  
554,  
555,  
556,  
557,  
558,  
559,  
560,  
561,  
562,  
563,  
564,  
565,  
566,  
567,  
568,  
569,  
570,  
571,  
572,  
573,  
574,  
575,

576,  
577,  
578,  
579,  
580,  
581,  
582,  
583,  
584,  
585,  
586,  
587,  
588,  
589,  
590,  
591,  
592,  
593,  
594,  
595,  
596,  
597,  
598,  
599,  
600,  
601,  
602,  
603,  
604,  
605,  
606,  
607,  
608,  
609,  
610,  
611,  
612,  
613,  
614,  
615,  
616,  
617,  
618,  
619,  
620,  
621,  
622,  
623,  
624,

625,  
626,  
627,  
628,  
629,  
630,  
631,  
632,  
633,  
634,  
635,  
636,  
637,  
638,  
639,  
640,  
641,  
642,  
643,  
644,  
645,  
646,  
647,  
648,  
649,  
650,  
651,  
652,  
653,  
654,  
655,  
656,  
657,  
658,  
659,  
660,  
661,  
662,  
663,  
664,  
665,  
666,  
667,  
668,  
669,  
670,  
671,  
672,  
673,

674,  
675,  
676,  
677,  
678,  
679,  
680,  
681,  
682,  
683,  
684,  
685,  
686,  
687,  
688,  
689,  
690,  
691,  
692,  
693,  
694,  
695,  
696,  
697,  
698,  
699,  
700,  
701,  
702,  
703,  
704,  
705,  
706,  
707,  
708,  
709,  
710,  
711,  
712,  
713,  
714,  
715,  
716,  
717,  
718,  
719,  
720,  
721,  
722,

723,  
724,  
725,  
726,  
727,  
728,  
729,  
730,  
731,  
732,  
733,  
734,  
735,  
736,  
737,  
738,  
739,  
740,  
741,  
742,  
743,  
744,  
745,  
746,  
747,  
748,  
749,  
750,  
751,  
752,  
753,  
754,  
755,  
756,  
757,  
758,  
759,  
760,  
761,  
762,  
763,  
764,  
765,  
766,  
767,  
768,  
769,  
770,  
771,

772,  
773,  
774,  
775,  
776,  
777,  
778,  
779,  
780,  
781,  
782,  
783,  
784,  
785,  
786,  
787,  
788,  
789,  
790,  
791,  
792,  
793,  
794,  
795,  
796,  
797,  
798,  
799,  
800,  
801,  
802,  
803,  
804,  
805,  
806,  
807,  
808,  
809,  
810,  
811,  
812,  
813,  
814,  
815,  
816,  
817,  
818,  
819,  
820,

821,  
822,  
823,  
824,  
825,  
826,  
827,  
828,  
829,  
830,  
831,  
832,  
833,  
834,  
835,  
836,  
837,  
838,  
839,  
840,  
841,  
842,  
843,  
844,  
845,  
846,  
847,  
848,  
849,  
850,  
851,  
852,  
853,  
854,  
855,  
856,  
857,  
858,  
859,  
860,  
861,  
862,  
863,  
864,  
865,  
866,  
867,  
868,  
869,



870,  
871,  
872,  
873,  
874,  
875,  
876,  
877,  
878,  
879,  
880,  
881,  
882,  
883,  
884,  
885,  
886,  
887,  
888,  
889,  
890,  
891,  
892,  
893,  
894,  
895,  
896,  
897,  
898,  
899,  
900,  
901,  
902,  
903,  
904,  
905,  
906,  
907,  
908,  
909,  
910,  
911,  
912,  
913,  
914,  
915,  
916,  
917,  
918,

919,  
920,  
921,  
922,  
923,  
924,  
925,  
926,  
927,  
928,  
929,  
930,  
931,  
932,  
933,  
934,  
935,  
936,  
937,  
938,  
939,  
940,  
941,  
942,  
943,  
944,  
945,  
946,  
947,  
948,  
949,  
950,  
951,  
952,  
953,  
954,  
955,  
956,  
957,  
958,  
959,  
960,  
961,  
962,  
963,  
964,  
965,  
966,  
967,

```
968,  
969,  
970,  
971,  
972,  
973,  
974,  
975,  
976,  
977,  
978,  
979,  
980,  
981,  
982,  
983,  
984,  
985,  
986,  
987,  
988,  
989,  
990,  
991,  
992,  
993,  
994,  
995,  
996,  
997,  
998,  
999,  
1000]
```

## Map functions

---

Now that you've been working a little bit with RDDs, let's make this a little more interesting. Imagine you're running a hot new e-commerce startup called BuyStuff, and you're trying to track of how much it charges customers from each item sold. In the next cell, we're going to create simulated data by multiplying the values 1-1000 with a random number from 0-1.

```
import numpy as np
```

```
np.random.seed(1)
```

```
nums = np.array(range(1, 1001))
sales_figures = nums * np.random.rand(1000)
sales_figures
```

```
array([4.17022005e-01, 1.44064899e+00, 3.43124452e-04, 1.20933029e+00,
       7.33779454e-01, 5.54031569e-01, 1.30382148e+00, 2.76448582e+00,
       3.57090727e+00, 5.38816734e+00, 4.61113966e+00, 8.22263400e+00,
       2.65787925e+00, 1.22936441e+01, 4.10813898e-01, 1.07274802e+01,
       7.09418164e+00, 1.00564169e+01, 2.66735183e+00, 3.96202978e+00,
       1.68156359e+01, 2.13017547e+01, 7.20875610e+00, 1.66157428e+01,
       2.19097288e+01, 2.32597733e+01, 2.29619371e+00, 1.09353393e+00,
       4.92508217e+00, 2.63442751e+01, 3.04875185e+00, 1.34754440e+01,
       3.16103545e+01, 1.81276197e+01, 2.42156990e+01, 1.13585627e+01,
       2.54005343e+01, 3.17157755e+01, 7.13242816e-01, 3.00057726e+01,
       4.05433046e+01, 3.14229575e+01, 1.20590917e+01, 3.47282905e+01,
       4.64517030e+00, 2.06031022e+01, 4.27039886e+01, 1.40934791e+01,
       1.41009916e+01, 6.50142861e+00, 9.87714851e-01, 3.52994477e+01,
       1.12162901e+01, 1.43395196e+01, 2.70365238e+01, 2.98830253e+00,
       3.27247035e+01, 8.51025734e+00, 3.47690267e+01, 4.19855016e+01,
       6.24240016e+00, 2.56714712e+01, 4.37472099e+01, 2.65074732e+01,
       3.24697483e+00, 3.53691628e+01, 4.44742412e+01, 3.50124596e+01,
       6.51770382e+01, 4.10588528e+01, 6.41415360e+01, 9.89817870e+00,
       1.01671733e+01, 5.97469554e+01, 2.98257628e+01, 1.25669190e+01,
       7.14181607e+01, 2.71257371e+01, 5.93141561e+01, 5.80798388e+01,
       7.15477934e+01, 5.11411210e+01, 6.23282220e+01, 2.93074607e+01,
       2.29438708e+01, 7.70462148e+01, 3.72439335e+01, 8.49059241e+01,
       5.90462933e+01, 5.59526148e+01, 1.04418835e+01, 8.73530118e+01,
       4.18418284e+01, 5.43686238e+01, 3.87729963e+01, 2.27545901e+01,
       8.76278135e+01, 5.62205897e+01, 2.84162376e-01, 6.17144914e+01,
       3.29911351e+01, 5.37599264e+01, 9.12520362e+01, 3.71560550e+01,
       9.53961908e+01, 6.60761723e+01, 1.69287298e+00, 1.00379221e+02,
       7.53077640e+01, 1.09705514e+02, 1.91297964e+01, 1.53592040e+01,
       1.05383287e+02, 7.94372704e+01, 7.59001986e+00, 8.76337141e+01,
       8.82035140e+01, 1.08916895e+02, 8.46714463e+01, 1.49125154e+01,
       2.40549619e+00, 3.19774040e+00, 3.48169803e+00, 3.05301724e+01,
       1.07503494e+02, 6.78927141e+01, 7.02083913e+01, 1.07779954e+02,
       1.60183577e+01, 3.62938783e+01, 7.67344646e+01, 1.27986639e+02,
       7.46170192e+01, 2.49873678e+00, 1.08085411e+02, 3.16845012e+01,
       1.10573412e+02, 5.35247689e+01, 1.20032318e+02, 1.04597030e+02,
       7.84298730e+01, 1.93766420e+01, 8.56822960e+00, 1.74734576e+01,
       6.46002239e+00, 1.56941428e+01, 3.31792728e+01, 1.05522369e+02,
       8.33978303e+01, 1.88339702e+00, 1.08681162e+01, 1.47026002e+02,
       8.69193707e+01, 3.13071581e+01, 3.91104904e+01, 1.16036833e+02,
       3.06824285e+01, 9.18547105e+01, 1.54233178e+02, 1.35492608e+02,
       3.86154892e+01, 7.99906937e+01, 1.01052782e+02, 1.35952868e+02,
       2.58705801e+01, 3.08364956e+00, 1.16936980e+01, 8.17059786e+01,
       1.02469679e+02, 9.67047443e+01, 5.42689720e+01, 1.70041979e+02,
       1.00295923e+02, 6.61445640e+01, 9.64159383e+01, 1.31178860e+02,
```

1.18454222e+02, 4.71556813e+01, 1.18739354e+01, 6.66151556e+01,  
1.13978869e+02, 3.82516698e+01, 1.37754266e+02, 1.22427126e+01,  
4.81582932e+01, 1.49684349e+02, 3.61722109e+01, 1.20218646e+02,  
9.91626884e+01, 1.75713514e+02, 5.02896832e+01, 1.26645294e+01,  
1.41867731e+02, 1.49802538e+02, 1.77024091e+02, 1.82666526e+02,  
2.74845988e+00, 4.64036931e+01, 1.22738893e+02, 1.89803264e+02,  
1.90985400e+02, 1.12443944e+02, 1.85868089e+02, 1.30879507e+02,  
7.99515814e+01, 1.00114077e+02, 1.25092270e+02, 1.14305968e+02,  
1.93571918e+02, 1.92934021e+02, 8.33187543e+01, 2.04211656e+02,  
3.70525570e+01, 2.70345172e+01, 2.90420190e+01, 1.09223028e+02,  
4.67088274e+00, 2.06657506e+02, 1.81138288e+02, 3.30417576e+00,  
3.89393725e+01, 7.37181135e+01, 2.92122964e+01, 1.81325915e+02,  
7.75657469e+01, 2.12464291e+02, 1.32117219e+02, 2.00373692e+02,  
1.93444188e+02, 2.08240233e+02, 1.06232341e+02, 1.26752461e+02,  
1.86074637e+02, 6.68582113e+01, 1.15209578e+02, 1.41390033e+02,  
3.68138631e+00, 1.41248575e+02, 1.03648647e+02, 1.93766527e+02,  
7.59739975e+01, 2.16079067e+02, 1.40419303e+02, 4.48984892e+01,  
1.93042662e+02, 1.50559670e+02, 1.33155902e+01, 1.04208033e+02,  
1.69088140e+02, 2.29650444e+02, 1.00908248e-01, 2.46143306e+02,  
9.52748196e+01, 2.47341019e+02, 1.54202606e+02, 2.12184527e+02,  
1.47700857e+02, 1.62043659e+02, 7.39642570e+01, 1.52576669e+02,  
1.95755680e+02, 2.24878225e+02, 1.98586616e+02, 1.84287114e+02,  
2.29087049e+02, 8.58331452e+01, 1.79100607e+02, 1.20834215e+02,  
1.02785640e+02, 1.10919064e+02, 1.08800967e+02, 8.63284333e+01,  
1.69783987e+02, 1.17887752e+02, 2.67795571e+02, 1.87073046e+02,  
5.50038591e+01, 1.18622881e+02, 9.57936009e+01, 2.23338865e+02,  
2.47279519e+02, 2.54883432e+02, 1.87549707e+02, 7.67391464e+01,  
7.19245099e+01, 2.44500812e+02, 1.51454103e+02, 2.31022392e+02,  
1.65449181e+02, 2.12611332e+02, 1.51032384e+02, 2.25098102e+02,  
1.66675391e+02, 1.36918704e+02, 1.01093228e+02, 2.01899671e+01,  
1.12243481e+02, 2.37285712e+01, 2.93862317e+02, 5.44838554e+01,  
2.44369468e+02, 2.64238417e+02, 2.08589215e+02, 1.73126301e+02,  
4.90962882e+01, 1.42865287e+02, 1.05967820e+02, 6.93123070e+01,  
1.83086167e+02, 9.68036497e+01, 2.84971027e+02, 2.83806284e+02,  
8.04780260e+01, 3.48198684e+01, 6.07832606e+01, 1.57868598e+02,  
2.30961657e+02, 6.62058314e+01, 7.91227051e+01, 2.72535000e+02,  
1.33487439e+02, 1.98572592e+02, 7.54741630e+01, 3.30373921e+01,  
1.67653531e+02, 1.55547962e+02, 4.99236276e+01, 2.03952444e+02,  
1.78979329e+02, 2.15865325e+02, 4.78445738e+01, 2.49507235e+02,  
7.39423636e+01, 1.73463509e+02, 2.63074169e+02, 7.50302381e+00,  
1.09310149e+02, 2.95047763e+02, 2.86356557e+02, 1.83069801e+02,  
2.95513421e+02, 3.24833649e+02, 2.83457600e+02, 2.93815713e+02,  
3.40664736e+01, 2.25351299e+02, 2.44120395e+02, 2.12363803e+02,  
2.79065726e+02, 1.20999270e+01, 2.70353796e+02, 2.57568467e+02,  
9.16735328e+01, 9.10025318e+01, 2.24467678e+02, 1.22925896e+02,  
2.84382158e+02, 1.59720351e+02, 2.81007040e+02, 3.56569842e+02,  
1.08389651e+02, 5.17681098e+01, 3.27174962e+02, 1.97127614e+02,  
3.55780235e+02, 2.32997210e+02, 3.64766080e+02, 2.00954056e+02,  
1.94251170e+02, 5.01083241e+01, 1.31966618e+02, 9.75330703e+00,

5.98274020e+01, 2.78868310e+02, 1.13998837e+01, 1.37820205e+02,  
3.25104537e+02, 2.61832177e+02, 2.61867072e+02, 7.16819844e+01,  
1.68365531e+02, 2.22162570e+02, 3.79074904e+02, 7.82999905e+01,  
9.53771672e+01, 1.01198810e+02, 2.90316724e+02, 1.77306427e+02,  
2.21455515e+01, 1.98321334e+02, 8.28764244e+01, 3.13052864e+02,  
1.16851233e+02, 1.08767687e+01, 2.34405818e+02, 3.34160810e+02,  
1.51263401e+02, 2.98443608e+02, 2.03945450e+02, 2.16380722e+02,  
3.84733163e+02, 3.23192278e+02, 1.30261958e+01, 2.86592449e+02,  
1.88325600e+02, 3.84704870e+02, 9.01231229e+01, 1.08965384e+02,  
3.33228516e+01, 1.75733720e+02, 4.48067120e+01, 2.61120145e+02,  
3.31623817e+02, 2.88475405e+02, 3.17977723e+02, 1.42460914e+02,  
3.52720068e+02, 1.79225348e+02, 3.45260136e+02, 2.63128387e+02,  
6.03811035e+01, 3.30792717e+01, 7.75470795e+00, 2.82913990e+01,  
1.94898121e+02, 4.82836591e+01, 1.18634900e+01, 3.23080714e+02,  
1.69390856e+02, 3.21183553e+02, 1.94986480e+02, 1.94437475e+02,  
2.07005395e+02, 2.05717704e+02, 3.49376054e+02, 1.75443139e+02,  
3.95347852e+02, 1.62327390e+01, 3.39730836e+02, 5.52822077e+01,  
2.72764483e+02, 4.58100351e+00, 2.38611886e+02, 1.33997274e+00,  
4.23281235e+02, 4.03809307e+02, 3.55797223e+02, 4.10042895e+02,  
6.53556458e+01, 7.09785314e+01, 8.46218845e+01, 2.81368148e+02,  
4.10331702e+02, 4.49439651e+02, 3.23560719e+02, 3.33700987e+02,  
4.15546994e+02, 1.83600169e+02, 1.14681462e+02, 7.97778786e+01,  
5.50697002e+01, 3.75426092e+02, 6.79648689e+01, 1.22634032e+02,  
3.80876468e+02, 1.44733659e+02, 4.58788949e+02, 1.24786913e+02,  
2.50283419e+02, 1.47799495e+02, 4.28974003e+02, 1.73014736e+02,  
2.05089171e+02, 2.42826736e+02, 4.45971077e+02, 1.47317270e+01,  
3.41951123e+02, 4.25907060e+02, 1.30705801e+01, 2.50584599e+02,  
1.56801099e+02, 4.14273853e+02, 2.69763494e+02, 3.34070288e+02,  
2.19633947e+02, 3.05358192e+02, 1.41277167e+02, 4.56210605e+00,  
2.82033652e+02, 1.52607665e+02, 2.53978391e+02, 4.50871680e+02,  
2.10252071e+02, 1.22213642e+02, 1.83790412e+02, 4.62203114e+02,  
4.65623586e+02, 4.20476314e+02, 4.59183051e+02, 1.13950145e+02,  
4.38285870e+01, 1.14109488e+02, 1.58131438e+02, 8.80820017e+01,  
3.06582552e+02, 2.09274726e+02, 4.13890216e+02, 9.40462423e+01,  
3.57255154e+02, 1.22581369e+02, 2.93425954e+02, 1.78681650e+02,  
2.92227369e+01, 1.17610227e+02, 3.42012816e+02, 2.56581048e+02,  
2.68331264e+02, 9.05050378e+01, 2.96201527e+02, 5.18311781e+02,  
4.25571092e+02, 3.10262510e+02, 5.10442283e+02, 4.72418794e+02,  
3.12694165e+02, 1.70562471e+01, 4.93151295e+01, 3.45162655e+01,  
2.38966836e+02, 1.98980462e+02, 5.17910868e+02, 8.93671120e+01,  
5.18495787e+02, 4.09831580e+02, 4.40967244e+02, 3.39082078e+02,  
3.59109498e+02, 2.56562695e+02, 7.08049638e+00, 1.90623286e+02,  
2.66210845e+02, 3.95709436e+02, 2.54465190e+02, 2.48828274e+02,  
7.50261938e+01, 5.94524866e+00, 4.14778208e+02, 1.75334155e+02,  
5.40426514e+02, 1.21128824e+02, 1.86628122e+02, 2.89190667e+02,  
4.17454976e+02, 2.56977212e+02, 6.92765114e+01, 1.73750766e+02,  
2.81017178e+02, 3.76007746e+02, 4.30513826e+02, 7.29880373e+01,  
1.28553887e+01, 2.91724265e+02, 4.56023644e+02, 7.10852460e+00,  
3.79945462e+02, 3.88733456e+02, 2.54722909e+02, 5.19599974e+02,

3.66641457e+02, 2.98670746e+00, 2.76608642e+02, 4.91529785e+02,  
4.75818954e+02, 3.72614509e+02, 3.87376594e+02, 3.33215758e+02,  
1.58167110e+02, 3.23986335e+02, 3.88931545e+02, 2.04409186e+02,  
4.97236281e+02, 1.13511817e+02, 4.35688032e+02, 1.69128004e+02,  
4.52672583e+02, 2.50654075e+02, 4.74118966e+02, 2.07878499e+02,  
1.25865317e+02, 4.52697860e+02, 1.82407391e+02, 4.34081080e+02,  
4.41472580e+02, 1.31509640e+02, 1.27396721e+02, 1.18572962e+02,  
8.50834476e+01, 2.25495394e+02, 1.59501029e+01, 6.65522214e+01,  
4.05412978e+02, 4.81465475e+02, 4.85593049e+01, 1.39948196e+02,  
1.25613525e+02, 5.55904139e+02, 4.31767913e+02, 3.36761844e+02,  
1.85451455e+02, 5.09260969e+02, 2.65971940e+02, 5.65155204e+02,  
4.32809756e+02, 2.93511223e+02, 7.76192138e+01, 6.01242827e+02,  
9.86173623e+01, 1.25208113e+02, 2.66901506e+02, 2.50605182e+02,  
9.11326696e+01, 4.53636368e+02, 1.17588177e+02, 4.01790882e+02,  
4.71441219e+02, 1.31918477e+02, 3.76798314e+02, 4.70327020e+02,  
4.01439569e+02, 3.76190201e+02, 1.86449322e+02, 4.62375290e+02,  
5.98380243e+02, 2.69805921e+02, 4.96685454e+02, 3.57057012e+01,  
5.32068011e+02, 1.22655511e+02, 2.52466899e+02, 1.92051869e+02,  
5.13464357e+01, 5.80773104e+02, 2.38009136e+02, 3.41769150e+02,  
3.18704992e+02, 8.53760977e+01, 1.33575776e+02, 4.93703483e+01,  
3.29641183e+02, 1.70007209e+02, 2.32447107e+02, 7.04585943e+01,  
5.14271351e+02, 6.97058554e+01, 6.45639280e+02, 1.16217724e+02,  
3.76070159e+02, 2.95082302e+01, 5.18709635e+02, 1.25139926e+02,  
3.48944530e+02, 4.89931334e+02, 9.94045745e+01, 3.65921884e+02,  
1.44050444e+02, 5.05624569e+02, 4.82184434e+02, 1.17934755e+02,  
5.76655627e+02, 1.32493167e+01, 5.77219025e+02, 3.75583361e+02,  
2.71367378e+02, 5.11395432e+02, 4.83927076e+02, 6.67432494e+02,  
1.88263578e+02, 2.57210989e+00, 6.34119871e+02, 5.83370031e+02,  
4.96347448e+02, 3.52381750e+02, 4.82851116e+02, 5.33882217e+02,  
2.56790021e+02, 5.28441252e+02, 5.15678907e+02, 4.21889314e+02,  
2.76885622e+02, 4.81142533e+02, 2.15098476e+00, 5.36228480e+02,  
6.21216706e+02, 1.66085101e+02, 8.39331931e+01, 1.53317656e+02,  
2.10561422e+02, 6.16353899e+02, 3.79673335e+02, 2.00698155e+02,  
9.69866375e+01, 2.03681413e+02, 4.31551376e+02, 2.28193525e+02,  
3.22438928e+02, 3.13546678e+02, 5.85491700e+02, 3.01854493e+02,  
2.45100464e+02, 4.79229839e+02, 1.57473742e+02, 3.32679027e+02,  
2.24427935e+02, 4.47574900e+02, 6.27312739e+02, 3.20545324e+02,  
5.62455969e+02, 3.28101373e+02, 4.71828890e+02, 9.49255020e+01,  
3.12179667e+02, 6.56523256e+02, 4.37761318e+02, 5.55144801e+02,  
3.65907941e+02, 3.61588384e+02, 6.12788184e+02, 4.93634482e+01,  
4.17915486e+02, 6.88216680e+02, 3.78555695e+02, 1.42348973e+02,  
6.21539575e+02, 1.84703130e+02, 5.15033637e+02, 3.97632052e+02,  
6.99292344e+02, 4.60760486e+02, 6.19265714e+02, 5.87032838e+00,  
7.33100982e+02, 5.76642112e+01, 2.39342229e+02, 7.03937382e+02,  
6.65963127e+00, 6.13756584e+02, 6.43325094e+02, 3.28993449e+02,  
1.91553150e+02, 6.02017140e+02, 3.58874365e+02, 1.01022634e+02,  
6.98670292e+02, 6.75561355e+02, 3.71116588e+02, 6.47667087e+02,  
3.16863566e+02, 5.18066394e+02, 3.02074895e+02, 3.84363933e+02,  
1.44248846e+02, 7.35321560e+02, 2.24486570e+02, 7.90431009e+01,

1.10401281e+02, 1.07946974e+01, 5.49130351e+02, 4.33534710e+02,  
6.11030752e+02, 3.90451541e+02, 6.10494021e+02, 5.37129973e+02,  
6.01276873e+02, 3.14617741e+02, 5.02022241e+02, 1.39520375e+02,  
2.50054112e+02, 1.34286394e+02, 3.18328398e+02, 1.88306623e+02,  
3.17806077e+02, 7.62623872e+02, 2.50810032e+02, 7.70272901e+02,  
4.99500309e+02, 2.94821540e+02, 6.74840301e+02, 4.88234351e+02,  
1.98854100e+02, 6.26355986e+02, 3.42454360e+02, 2.83148845e+02,  
2.61909611e+02, 5.53710887e+02, 2.13576849e+02, 6.43389299e+02,  
2.35345169e+02, 4.34208864e+02, 3.89849272e+02, 6.84285130e+02,  
7.11597534e+02, 1.47876323e+02, 4.70034813e+02, 7.22156857e+02,  
3.59124361e+02, 7.43025854e+02, 2.25145649e+02, 4.91935589e+02,  
5.52105051e+02, 1.84846644e+02, 1.11654527e+01, 3.38379857e+02,  
7.62985778e+02, 2.79224881e+02, 6.35491601e+02, 1.42584831e+02,  
2.79375467e+02, 1.18280936e+02, 5.87035297e+02, 5.73432251e+02,  
5.65256298e+02, 2.08291540e+02, 5.69812380e+02, 1.87293174e+02,  
3.50335521e+02, 3.07207672e+02, 2.93839643e+02, 4.77381804e+01,  
5.23635054e+02, 5.87072780e+02, 5.09892205e+02, 5.39396213e+02,  
1.41560615e+02, 1.24638630e+02, 4.29336161e+02, 7.31778159e+02,  
1.53969031e+02, 3.87859181e+02, 3.59874228e+02, 4.17722916e+02,  
1.35830558e+02, 2.88335020e+02, 2.20765181e+02, 7.12780740e+02,  
6.76280748e+02, 3.60936728e+02, 5.14142095e+02, 1.23354838e+02,  
4.32661687e+02, 2.52405023e+02, 7.31562961e+02, 5.72201821e+02,  
5.40353333e+02, 1.06537610e+02, 4.02352637e+02, 8.44506299e+02,  
8.12692392e+02, 5.53483479e+02, 1.30331664e+02, 5.49649090e+02,  
4.87035079e+02, 4.03989952e+02, 3.69396336e+02, 5.17769069e+02,  
7.35223954e+02, 6.50470821e+02, 5.02305588e+02, 8.02643230e+02,  
5.62590456e+01, 8.62471463e+02, 4.61582467e+01, 1.73960116e+02,  
3.69063074e+02, 9.39627578e+01, 5.45711610e+02, 4.20414861e+01,  
2.49615144e+02, 5.35902060e+01, 6.18393492e+02, 5.88241427e+02,  
3.33529503e+02, 1.65987337e+02, 6.59605264e+02, 3.00895293e+02,  
7.03841533e+02, 4.32280232e+02, 4.66268797e+02, 2.52998743e+01,  
5.72722266e+02, 3.12084287e+02, 2.04221679e+02, 3.87023972e+02,  
3.41543439e+02, 4.19991402e+02, 8.76637602e+02, 3.26482751e+02,  
6.94645890e+02, 4.96385302e+02, 7.99328849e+02, 3.19457566e+02,  
2.21212331e+02, 8.21739354e+02, 3.93114402e+01, 8.59481099e+02,  
5.03548259e+02, 3.40985074e+02, 9.02512456e+02, 5.29932827e+01,  
4.69686081e+02, 2.82983387e+01, 5.20341098e+02, 1.64587280e+02,  
5.76065750e+02, 8.96564240e+02, 8.00535834e+02, 4.13882004e+02,  
6.49658618e+02, 7.13716313e+02, 4.54760794e+02, 4.86250811e+02,  
1.38872431e+02, 3.40586731e+02, 1.31270217e+02, 6.71649828e+02,  
4.41237020e+02, 4.15661803e+02, 8.21320119e+02, 4.89630221e+02,  
3.80045383e+02, 2.50069581e+02, 6.70432047e+01, 3.89702877e+02,  
2.40279991e+01, 2.71937780e+02, 4.70781392e+02, 9.04113349e+02,  
1.02491791e+02, 6.31312509e+02, 4.69436496e+02, 7.30472336e+02,  
1.35134185e+02, 7.83768835e+01, 3.76463145e+02, 7.52332366e+02,  
1.81133573e+02, 7.26317224e+02, 2.74912203e+02, 2.05613130e+02,  
1.58630940e+01, 3.78726107e+02, 3.62408498e+02, 6.27696379e+02,  
6.75852356e+01, 1.45584204e+02, 1.58299682e+01, 1.08789317e+02,  
6.23762337e+02, 3.85745264e+02, 3.07864235e+02, 5.35595514e+02,



```
9.54715505e+02, 8.02776026e+02, 6.73737125e+02, 8.85201280e+02,
3.83382019e+01, 6.79421046e+01, 4.58364082e+02, 3.37994056e+02,
9.08197248e+02, 4.74878012e+02, 5.23999286e+02, 8.70193075e+02,
4.34575900e+02, 8.54231495e+02, 2.47242206e+02, 2.67238271e+02,
3.20809078e+02, 5.35517857e+02, 2.15505972e+02, 6.58000588e+02,
1.40080210e+02, 9.24064696e+01, 8.55398487e+02, 2.33078804e+02,
3.80213951e+02, 5.63540503e+02, 5.18966544e+02, 7.51115929e+01,
8.64510550e+02, 9.41624262e+02, 8.05194738e+02, 2.81531420e+02,
5.24151869e+02, 3.37380224e+02, 5.51893974e+02, 9.70505855e+02,
3.10767809e+02, 6.67459013e+02, 3.25641240e+02, 7.74477266e+02])
```

We now have sales prices for 1000 items currently for sale at BuyStuff. Now create an RDD called `price_items` using the newly created data with 10 slices. After you create it, use one of the basic actions to see what's in the RDD.

```
price_items = sc.parallelize(sales_figures, numSlices=10)
price_items.take(4)
```

```
[0.417022004702574,
1.4406489868843162,
0.0003431244520346599,
1.209330290527359]
```

Now let's perform some operations on this simple dataset. To begin with, create a function that will take into account how much money BuyStuff will receive after sales tax has been applied. Assume a sales tax of 8%, i.e. that the current price data is actually 108% of what it should be.

To make this happen, create a function called `sales_tax()` that returns the amount of money our company will receive after the sales tax has been applied. The function will have this parameter:

- `num` : (float) number to be multiplied by the sales tax.

Apply that function to the RDD by using the `.map()` method and assign it to a variable `revenue_minus_tax`

```
def sales_tax(num):
    return num / 1.08

revenue_minus_tax = price_items.map(sales_tax)
```

Remember, Spark has **lazy evaluation**, which means that the `sales_tax()` function is a transformer that is not executed until you call an action. Use one of the collection methods to execute the transformer now a part of the RDD and observe the contents of the `revenue_minus_tax` RDD.

```
revenue_minus_tax.take(10)
```

```
[0.38613148583571666,  
 1.3339342471151074,  
 0.0003177078259580184,  
 1.119750269006814,  
 0.6794254204495974,  
 0.5129921931599878,  
 1.2072421107812004,  
 2.559709089207761,  
 3.306395618588916,  
 4.989043833364416]
```

## Lambda Functions

Note that you can also use lambda functions if you want to quickly perform simple operations on data without creating a function. Let's assume that BuyStuff has also decided to offer a 10% discount on all of their items on the pre-tax amounts of each item. Use a lambda function within a `.map()` method to apply the additional 10% loss in revenue for BuyStuff and assign the transformed RDD to a new RDD called `discounted`.

```
discounted = revenue_minus_tax.map(lambda x : x*0.9)
```

```
discounted.take(10)
```

```
[0.347518337252145,  
 1.2005408224035967,  
 0.0002859370433622166,  
 1.0077752421061326,  
 0.6114828784046377,  
 0.461692973843989,  
 1.0865178997030804,  
 2.303738180286985,
```

```
2.9757560567300243,  
4.490139450027975]
```

## Chaining Methods

---

You are also able to chain methods together with Spark. In one line, remove the tax and discount from the revenue of BuyStuff and use a collection method to see the 15 costliest items.

```
price_items.map(sales_tax).map(lambda x : x*0.9).top(15)
```

```
[808.7548791977921,  
795.5962541991298,  
784.6868851954209,  
756.8310401687224,  
753.4277907153834,  
752.0937136346447,  
747.1368663633596,  
737.66773326942,  
730.5313349924388,  
725.1608962422889,  
720.4254585887082,  
718.7262188305615,  
716.2342491392883,  
712.8320720999051,  
711.8595792911022]
```

## RDD Lineage

---

We are able to see the full lineage of all the operations that have been performed on an RDD by using the `RDD.debugString()` method. As your transformations become more complex, you are encouraged to call this method to get a better understanding of the dependencies between RDDs. Try calling it on the `discounted` RDD to see what RDDs it is dependent on.

```
discounted.debugString()
```

```
b'(10) PythonRDD[10] at RDD at PythonRDD.scala:53 []\n |
ParallelCollectionRDD[5] at readRDDFromFile at PythonRDD.scala:274 []'
```

## Map vs. Flatmap

Depending on how you want your data to be outputted, you might want to use `.flatMap()` rather than a simple `.map()`. Let's take a look at how it performs operations versus the standard map. Let's say we wanted to maintain the original amount BuyStuff receives for each item as well as the new amount after the tax and discount are applied. Create a map function that will return a tuple with (original price, post-discount price).

```
mapped = price_items.map(lambda x: (x, x / 1.08 * 0.9))
print(mapped.count())
print(mapped.take(10))
```

```
1000
[(0.417022004702574, 0.347518337252145), (1.4406489868843162,
1.2005408224035967), (0.0003431244520346599, 0.0002859370433622166),
(1.209330290527359, 1.0077752421061326), (0.7337794540855652,
0.6114828784046377), (0.5540315686127868, 0.461692973843989),
(1.3038214796436964, 1.0865178997030804), (2.764485816344382, 2.303738180286985),
(3.5709072680760294, 2.9757560567300243), (5.3881673400335695,
4.490139450027975)]
```

Note that we have 1000 tuples created to our specification. Let's take a look at how `.flatMap()` differs in its implementation. Use the `.flatMap()` method with the same function you created above.

```
flat_mapped = price_items.flatMap(lambda x : (x, x / 1.08 * 0.9 ))
print(flat_mapped.count())
print(flat_mapped.take(10))
```

```
2000
[0.417022004702574, 0.347518337252145, 1.4406489868843162, 1.2005408224035967,
0.0003431244520346599, 0.0002859370433622166, 1.209330290527359,
1.0077752421061326, 0.7337794540855652, 0.6114828784046377]
```

Rather than being represented by tuples, all of the values are now on the same level. When we are trying to combine different items together, it is sometimes necessary to use `.flatMap()` rather than `.map()` in order to properly reduce to our specifications. This is not one of those instances, but in the upcoming lab, you just might have to use it.

## Filter

---

After meeting with some external consultants, BuyStuff has determined that its business will be more profitable if it focuses on higher ticket items. Now, use the `.filter()` method to select items that bring in more than \$300 after tax and discount have been removed. A filter method is a specialized form of a map function that only returns the items that match a certain criterion. In the cell below:

- use a lambda function within a `.filter()` method to meet the consultant's suggestion's specifications. `set RDD = selected_items`
- calculate the total number of items remaining in BuyStuff's inventory

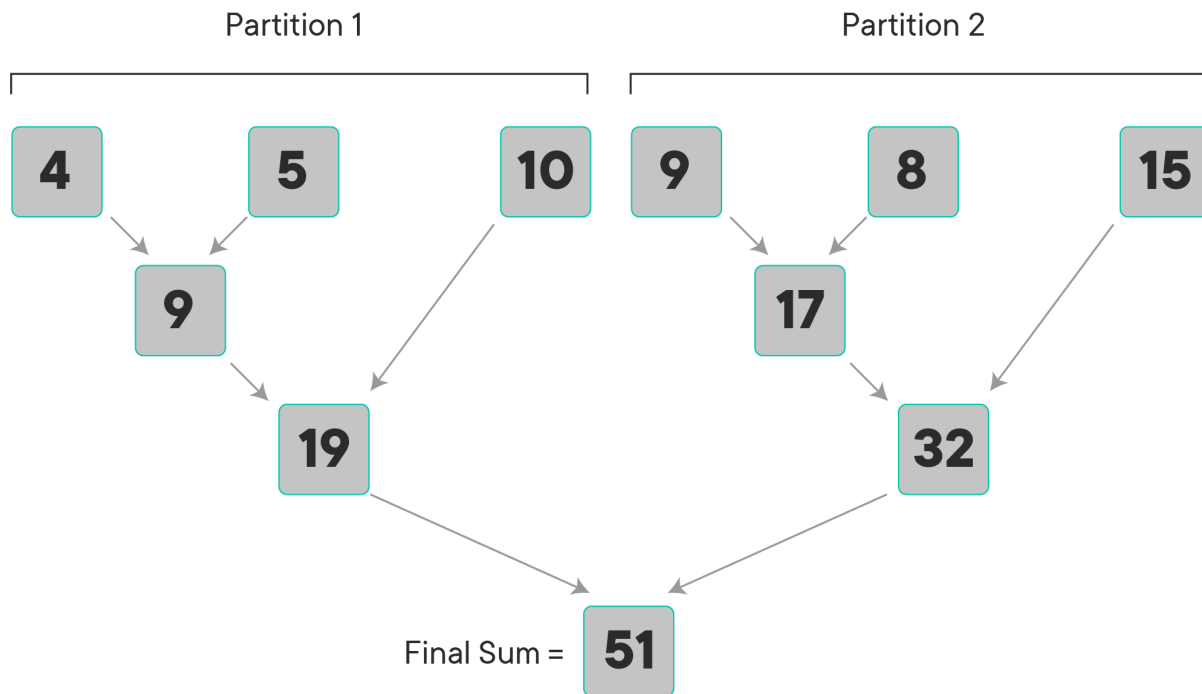
```
selected_items = discounted.filter(lambda x: x > 300)
selected_items.count()
```

272

## Reduce

---

Reduce functions are where you are in some way combining all of the variables that you have mapped out. Here is an example of how a reduce function works when the task is to sum all values:



As you can see, the operation is performed within each partition first, after which, the results of the computations in each partition are combined to come up with one final answer.

Now it's time to figure out how much money BuyStuff would make from selling one of all of its items after they've reduced their inventory. Use the `.reduce()` method with a lambda function to add up all of the values in the RDD. Your lambda function should have two variables.

```
selected_items.reduce(lambda x, y: x + y)
```

```
123432.4660522227
```

The time has come for BuyStuff to open up shop and start selling its goods. It only has one of each item, but it's allowing 50 lucky users to buy as many items as they want while they remain in stock. Within seconds, BuyStuff is sold out. Below, you'll find the sales data in an RDD with tuples of (user, item bought).

```
import random
random.seed(42)
# generating simulated users that have bought each item
sales_data = selected_items.map(lambda x: (random.randint(1, 50), x))

sales_data.take(7)
```

```
[(8, 303.9717333600381),  
 (36, 315.8957533626659),  
 (29, 320.6109689548023),  
 (18, 320.58739183867544),  
 (48, 329.4565433918027),  
 (36, 352.7343621192842),  
 (4, 336.5077558595325)]
```

It's time to determine some basic statistics about BuyStuff users.

Let's start off by creating an RDD that determines how much each user spent in total. To do this we can use a method called `.reduceByKey()` to perform reducing operations while grouping by keys. After you have calculated the total, use the `.sortBy()` method on the RDD to rank the users from the highest spending to the least spending.

```
total_spent = sales_data.reduceByKey(lambda x, y: x + y)  
total_spent.take(10)
```

```
[(10, 2505.4128261985734),  
 (30, 3382.449750500504),  
 (20, 2162.532818544579),  
 (40, 2733.835414659196),  
 (50, 1009.1271615645499),  
 (41, 2288.6328380275563),  
 (21, 1511.275378505312),  
 (1, 2752.542147304854),  
 (31, 3056.504120650862),  
 (11, 1511.8044372912896)]
```

```
total_spent.sortBy(lambda x: x[1], ascending = False).collect()
```

```
[(37, 6058.32282961381),  
 (35, 5147.3165498027965),  
 (14, 4718.853598241101),  
 (3, 4577.11556237689),  
 (32, 3661.5914194018924),  
 (39, 3590.199456355873),  
 (34, 3584.8861644479093),  
 (4, 3450.7117495281045),  
 (30, 3382.449750500504),  
 (42, 3314.794549097422),  
 (19, 3258.7620596071074),
```

```
(22, 3200.1246013611844),  
(31, 3056.504120650862),  
(36, 3053.623359721479),  
(45, 3010.218962331668),  
(8, 2912.0427078664316),  
(46, 2837.3218299486853),  
(44, 2832.057231700557),  
(1, 2752.542147304854),  
(40, 2733.835414659196),  
(13, 2699.181298186921),  
(26, 2663.0870131901474),  
(2, 2634.2461944306783),  
(10, 2505.4128261985734),  
(17, 2474.19078593852),  
(12, 2380.99877057044),  
(41, 2288.6328380275563),  
(29, 2271.601948461445),  
(49, 2206.520353564143),  
(20, 2162.532818544579),  
(16, 2070.5342988892085),  
(5, 1988.2251520732625),  
(24, 1926.9060030801184),  
(33, 1899.7417139575373),  
(48, 1717.7722170913798),  
(43, 1653.3750643830203),  
(28, 1639.9325449739288),  
(23, 1593.6702466360014),  
(11, 1511.8044372912896),  
(21, 1511.275378505312),  
(15, 1497.4823720659315),  
(18, 1472.6619623487884),  
(47, 1451.1062895771497),  
(6, 1413.623445906109),  
(25, 1097.048421376119),  
(50, 1009.1271615645499),  
(7, 905.4233605392121),  
(38, 837.0751484844222),  
(9, 816.0019218480354)]
```

Next, let's determine how many items were bought per user. This can be solved in one line using an RDD method. After you've counted the total number of items bought per person, sort the users from most number of items bought to least number of items. Time to start a customer loyalty program!

```
sorted(sales_data.countByKey().items(), key=lambda kv:kv[1], reverse=True)
```



```
[(49, 11),
 (22, 11),
 (11, 10),
 (23, 10),
 (28, 9),
 (13, 8),
 (38, 8),
 (44, 8),
 (18, 8),
 (35, 8),
 (48, 8),
 (2, 7),
 (43, 7),
 (46, 7),
 (37, 7),
 (20, 7),
 (32, 7),
 (47, 7),
 (17, 7),
 (8, 6),
 (40, 6),
 (16, 5),
 (4, 5),
 (33, 5),
 (6, 5),
 (26, 5),
 (34, 5),
 (5, 5),
 (36, 5),
 (30, 4),
 (24, 4),
 (42, 4),
 (9, 4),
 (31, 4),
 (19, 4),
 (27, 4),
 (25, 4),
 (21, 3),
 (10, 3),
 (50, 3),
 (41, 3),
 (3, 3),
 (7, 3),
 (29, 3),
 (14, 2),
 (1, 2),
 (45, 2),
 (12, 2),
```

```
(39, 2),  
(15, 2)]
```

## Stop the SparkContext

Now that we are finished with our analysis, stop the `sc`.

```
sc.stop()
```

## Additional Reading

- [The original paper on RDDs](#)
- [RDDs in Apache Spark](#)
- [Programming with RDDs](#)
- [RDD Transformations and Actions Summary](#)

## Summary

In this lab we went through a brief introduction to RDD creation from a Python collection, setting a number of logical partitions for an RDD and extracting lineage. We also used transformations and actions to perform calculations across RDDs on a distributed setup. In the next lab, you'll get the chance to apply these transformations on different books to calculate word counts and various statistics.

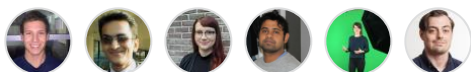
## Releases

No releases published

## Packages

No packages published

## Contributors 6



## Languages

- Jupyter Notebook 100.0%