```
Let's use the Docker official image
        docker pull spark:python3
We run the Docker container and mount the Docker volume with some
input data (we assume that the volume contains the input file
input.txt)
        docker run -it -v $PWD/data:/data -p 8080:8080 -p:4040:4040
--rm spark:python3 /opt/spark/bin/pyspark

Let's play with Spark using pyspark, the interactive shell in
Python, already running in the container
PySparkShell application UI is available at http://localhost:4040


--- Create RDDs

lines = sc.textFile("/data/input.txt")
lines.collect()

--- Transformations

# map: transform each element through a function
nums = sc.parallelize([1, 2, 3, 4])
squares = nums.map(lambda x: x * x)
nums.collect()
squares.collect()

# filter: select those elements that func returns true
even = squares.filter(lambda num: num % 2 == 0)
even.collect()

# flatMap: map each element to zero or more others
ranges = nums.flatMap(lambda x: range(0, x, 1))
ranges.collect()
# splitting input lines into words
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.collect()

#union: return the union of this RDD and another one
rdd1 = sc.parallelize([2, 4, 7, 9])
rdd2 = sc.parallelize([1, 4, 5, 8, 9])
rdd3 = rdd1.union(rdd2)
rdd3.collect()

#distinct: return a new RDD that contains the distinct elements
#of the source RDD: helpful to remove duplicate data
rdd3.distinct().collect()

#reduceByKey: aggregate values with identical key
x = sc.parallelize([("a", 1), ("b", 1), ("a", 1), ("a", 1), ("b",
1), ("b", 1), ("b", 1), ("b", 1)], 3)
y = x.reduceByKey(lambda accum, n: accum + n)
y.collect()
```

```
#join: perform equi-join on the keys of two RDDs
users = sc.parallelize([(0, "Alex"), (1, "Bert"), (2, "Curt"), (3,
"Don")])
hobbies = sc.parallelize([(0, "writing"), (0, "gym"), (1,
"swimming")])
users.join(hobbies).collect()
#Let's analyze the DAG using the Spark Web UI

#groupByKey: group the values for each key in the RDD into a single
#sequence. Hash-partitions the resulting RDD with numPartitions
#partitions.
#If you are grouping in order to perform an aggregation
#(e.g., sum or average) over each key, using reduceByKey or
#aggregateByKey will provide much better performance.
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
sorted(rdd.groupByKey().mapValues(len).collect())

#mapValues: 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.
#Differently from map, it operates on the value only.
fruits = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b",
["grapes"])])
fruits.mapValues(len).collect()

#mapPartitions: return a new RDD by applying a function to each
partition of this RDD
rdd = sc.parallelize([1, 2, 3, 4], 2)
def f(iterator): yield sum(iterator)
rdd.mapPartitions(f).collect()
[3, 7]


--- Actions

nums = sc.parallelize([1, 2, 3, 4])
nums.collect()

nums.take(3)

nums.count()

sum = nums.reduce(lambda x, y: x + y)
sum

nums.saveAsTextFile("/data/out")


--- Greek Pi calculation using Monte Carlo method

import random
NUM_SAMPLES = 100000

def inside(p):
        x, y = random.random(), random.random()
        return x*x + y*y < 1

samples = sc.parallelize(range(0, NUM_SAMPLES))
within_circle = samples.filter(inside)
count = within_circle.count()
print("Pi is roughly %f" % (4.0 * count / NUM_SAMPLES))


--- WordCount

text_file = sc.textFile("/data/input.txt")

counts = text_file.flatMap(lambda line: line.split(" ")) \
        .map(lambda word: (word, 1)) \
        .reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile("/data/output")


--- WordCount using countByValue: when can we use it?

text_file = sc.textFile("/data/input.txt")

counts = text_file.flatMap(lambda line: line.split(" "))
wordCount = words.countByValue()
print(wordCount)


--- Compute mean value

# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31),
("Jules", 30), ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average
agesRDD = (dataRDD
.map(lambda x: (x[0], (x[1], 1)))
.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
.map(lambda x: (x[0], x[1][0]/x[1][1])))

# Let's examine the code snippet step by step
mapRDD = dataRDD.map(lambda x: (x[0], (x[1], 1)))
mapRDD.collect()
redRDD = mapRDD.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
redRDD.collect()
agesRDD = redRDD.map(lambda x: (x[0], x[1][0]/x[1][1]))
agesRDD.collect()


##############

# Launch applications on Spark
```

```
docker run -it -p 8080:8080 -p:4040:4040 --rm spark:python3 /bin/
bash

# Inside the container
cd ..

# Submit PageRank application in Python
./bin/spark-submit examples/src/main/python/pagerank.py data/mllib/
pagerank_data.txt 10

# Submit Pi estimation application in Java
./bin/spark-submit --class org.apache.spark.examples.SparkPi \
    --master local \
    --deploy-mode client \
    --num-executors 2 \
    --driver-memory 512m \
    --executor-memory 512m \
    --executor-cores 1 \
    examples/jars/spark-examples*.jar 100
```