Let's use a Docker container provided by Bitnami (https://github.com/bitnami/bitnami-docker-spark#how-to-use-this-image).

```
docker pull bitnami/spark:latest
```

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 -i -t -p 8080:8080 -p:4040:4040 --name spark -v
$PWD/data:/data bitnami/spark /bin/sh
```

Let's play with Spark using the interactive shell in Python. To run it:

```
pyspark
```

PySparkShell application UI is available at http://localhost:4040

--- Create RDDs

```
lines = sc.textFile("input.txt")
lines.collect()
```

--- Transformations

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

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

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

#reduceByKey: aggregates 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)
```

```python
y.collect()

#join: performs 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: groups 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: passes 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()


--- 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("file.txt")


--- 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("input.txt")

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

counts.saveAsTextFile("output")
```