STATS 507
Data Analysis in Python

Lecture 18: PySpark

Some slides adapted from C. Budak and R. Burns

Parallel Computing with Apache Spark

Apache Spark is a computing framework for large-scale parallel processing

Developed by UC Berkeley AMPLab (Now RISELab)

now maintained by Apache Foundation

Implementations are available in Java, Scala and Python (and R, sort of)

and these can be run interactively!

Easily communicates with several other “big data” Apache tools

e.g., Hadoop, Mesos, HBase

Can also be run locally or in the cloud (e.g., Amazon EC2)

https://spark.apache.org/docs/0.9.0/index.html

Why use Spark?

vs

“Wait, doesn’t Hadoop/mrjob already do all this stuff?”

Short answer: yes!

Less short answer: Spark is faster and more flexible than Hadoop

and since Spark looks to be eclipsing Hadoop in industry, it is my responsibility to teach it to you

Spark still follows the MapReduce framework, but is better suited to:

- Interactive sessions
- Caching (i.e., data is stored in RAM on the nodes where it is to be processed, not on disk)
- Repeatedly updating computations (e.g., updates as new data arrive)
- Fault tolerance and recovery

Apache Spark: Overview

Implemented in Scala

- Popular functional programming (sort of…) language
- Runs atop Java Virtual Machine (JVM)
- http://www.scala-lang.org/

But Spark can be called from Scala, Java and Python

- and from R using SparkR: https://spark.apache.org/docs/latest/sparkr.html

We'll do all our coding in Python

- PySpark: https://spark.apache.org/docs/0.9.0/python-programming-guide.html
- but everything you learn can be applied with minimal changes in other supported languages

Running Spark

Option 1: Run in interactive mode

- Type `pyspark` on the command line
- PySpark provides an interface similar to the Python interpreter
- Scala, Java and R also provide their own interactive modes

Option 2: Run on a cluster

- Write your code, then launch it via a scheduler
- `spark-submit`
  - https://spark.apache.org/docs/latest/submitting-applications.html#launching-applications-with-spark-submit
  - http://arc-ts.umich.edu/hadoop-user-guide/#document-5
- Similar to running Python `mrjob` scripts with the `-r hadoop` flag

Two Basic Concepts

SparkContext

- Object corresponding to a connection to a Spark cluster
  - Automatically created in interactive mode
  - Must be created explicitly when run via scheduler (We'll see an example soon)
- Maintains information about where data is stored
- Allows configuration by supplying a `SparkConf` object

Resilient Distributed Dataset (RDD)

- Represents a collection of data
- Distributed across nodes in a fault-tolerant way (much like HDFS)

More about RDDs

RDDs are the basic unit of Spark

“a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.” (https://spark.apache.org/docs/0.9.0/scala-programming-guide.html#overview)

Elements of an RDD are analogous to <key,value> pairs in MapReduce

RDD is roughly analogous to a dataframe in R

RDD elements are somewhat like rows in a table

Spark can also keep (**persist**, in Spark’s terminology) an RDD in memory

Allows reuse or additional processing later

RDDs are **immutable**, like Python tuples and strings.

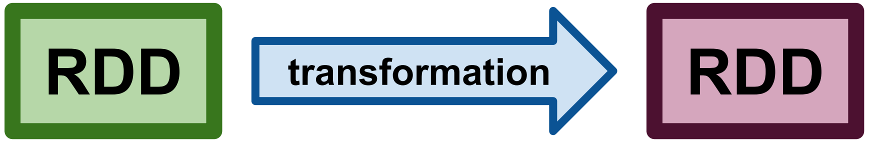
RDD operations

Think of RDD as representing a data set

Two basic operations:

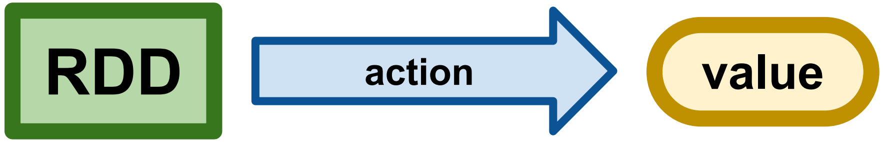
Transformation: results in another RDD

(e.g., `map` takes an RDD and applies some function to every element of the RDD)



Action: computes a value and reports it to driver program

(e.g., `reduce` takes all elements and computes some summary statistic)



RDD operations are lazy!

Transformations are only carried out once an **action** needs to be computed.

Spark remembers the sequence of transformations to run...

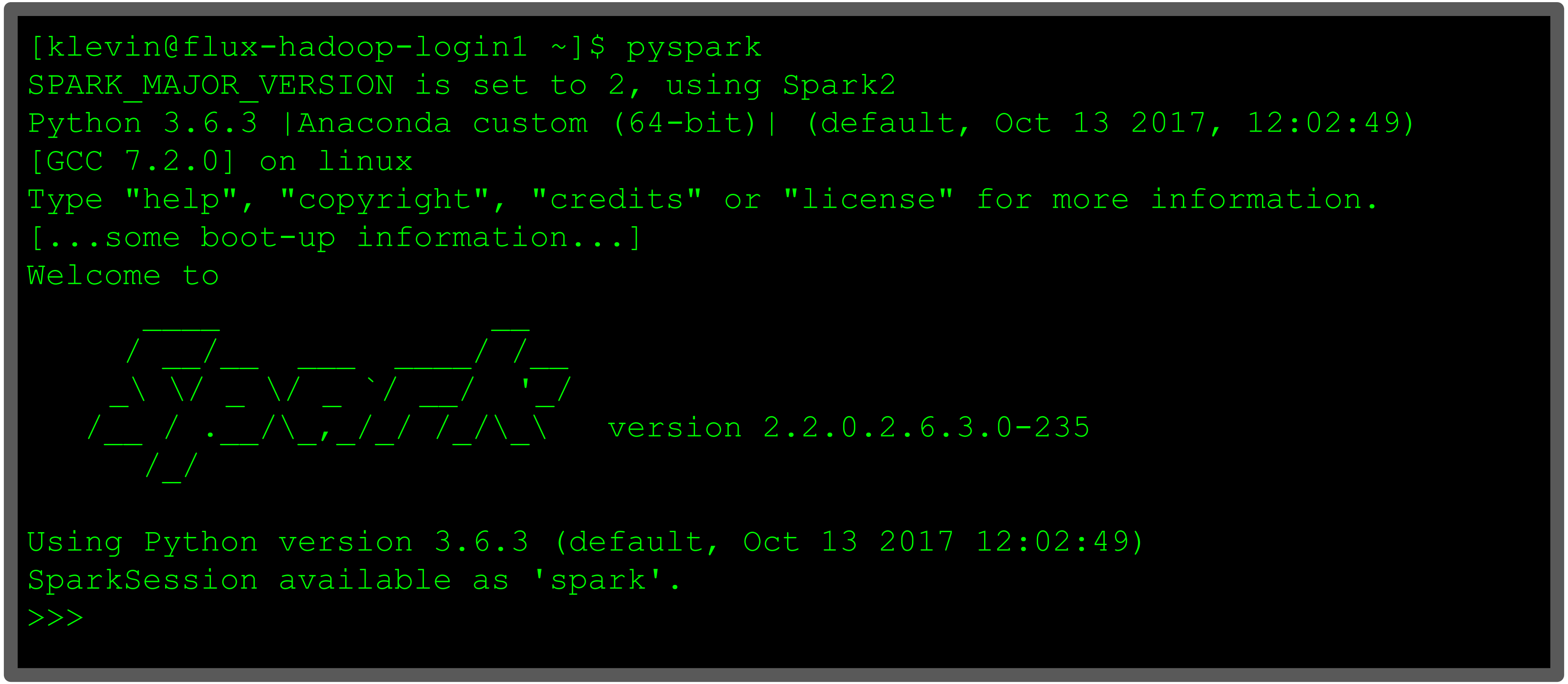
...but doesn't execute them until it has to

e.g., to produce the result of a reduce operation for the user.

This allows for gains in efficiency in some contexts

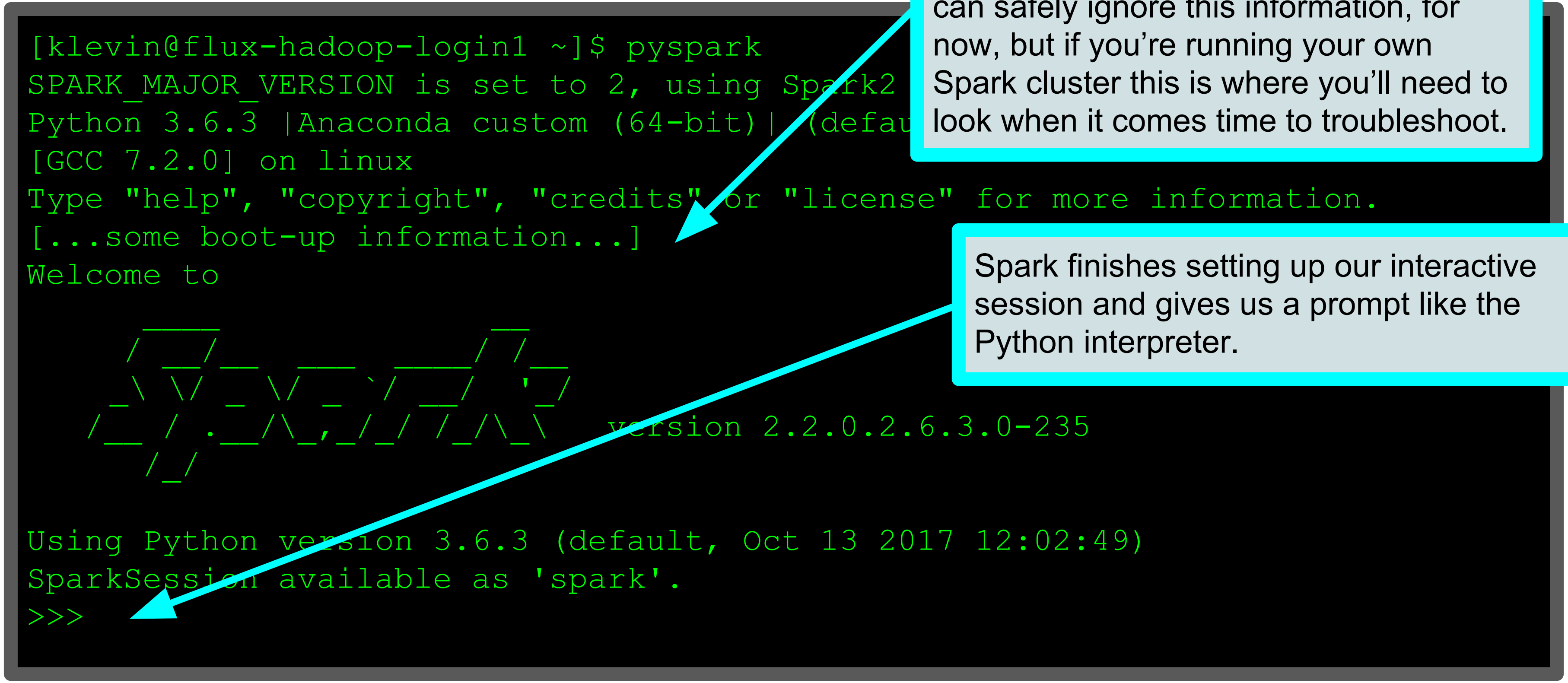
mainly because it avoids expensive intermediate computations

Okay, let’s dive in!



```
[klevin@flux-hadoop-login1 ~]$ pyspark
SPARK_MAJOR_VERSION is set to 2, using Spark2
Python 3.6.3 |Anaconda custom (64-bit)| (default, Oct 13 2017, 12:02:49)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
[...some boot-up information...]
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0.2.6.3.0-235
      /_/

Using Python version 3.6.3 (default, Oct 13 2017 12:02:49)
SparkSession available as 'spark'.
>>>
```

Okay, let's dive in!

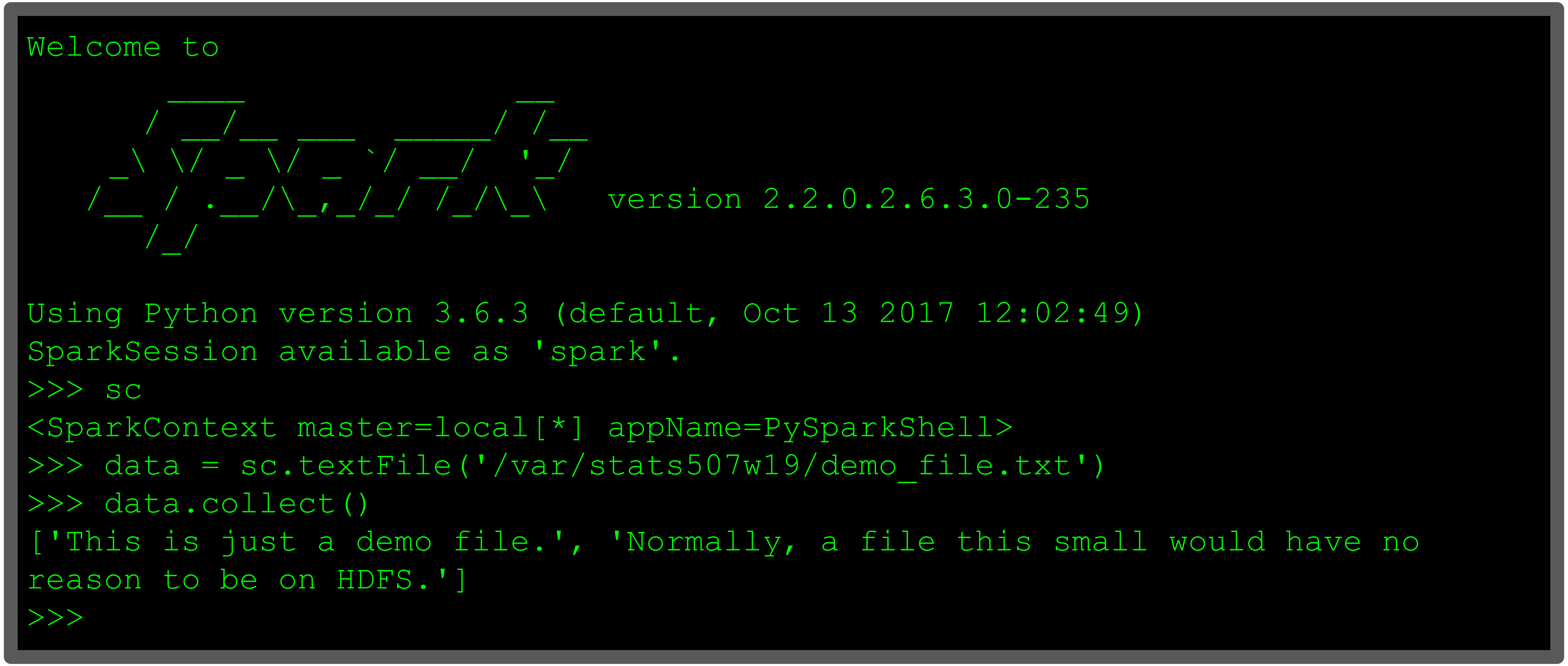


```
[klevin@flux-hadoop-login1 ~]$ pyspark
SPARK_MAJOR_VERSION is set to 2, using Spark2
Python 3.6.3 |Anaconda custom (64-bit)| (defau
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
[...some boot-up information...]
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0.2.6.3.0-235
      /_/

Using Python version 3.6.3 (default, Oct 13 2017 12:02:49)
SparkSession available as 'spark'.
>>>
```

There will be information here (sometimes multiple screens' worth) about establishing a Spark session. You can safely ignore this information, for now, but if you're running your own Spark cluster this is where you'll need to look when it comes time to troubleshoot.

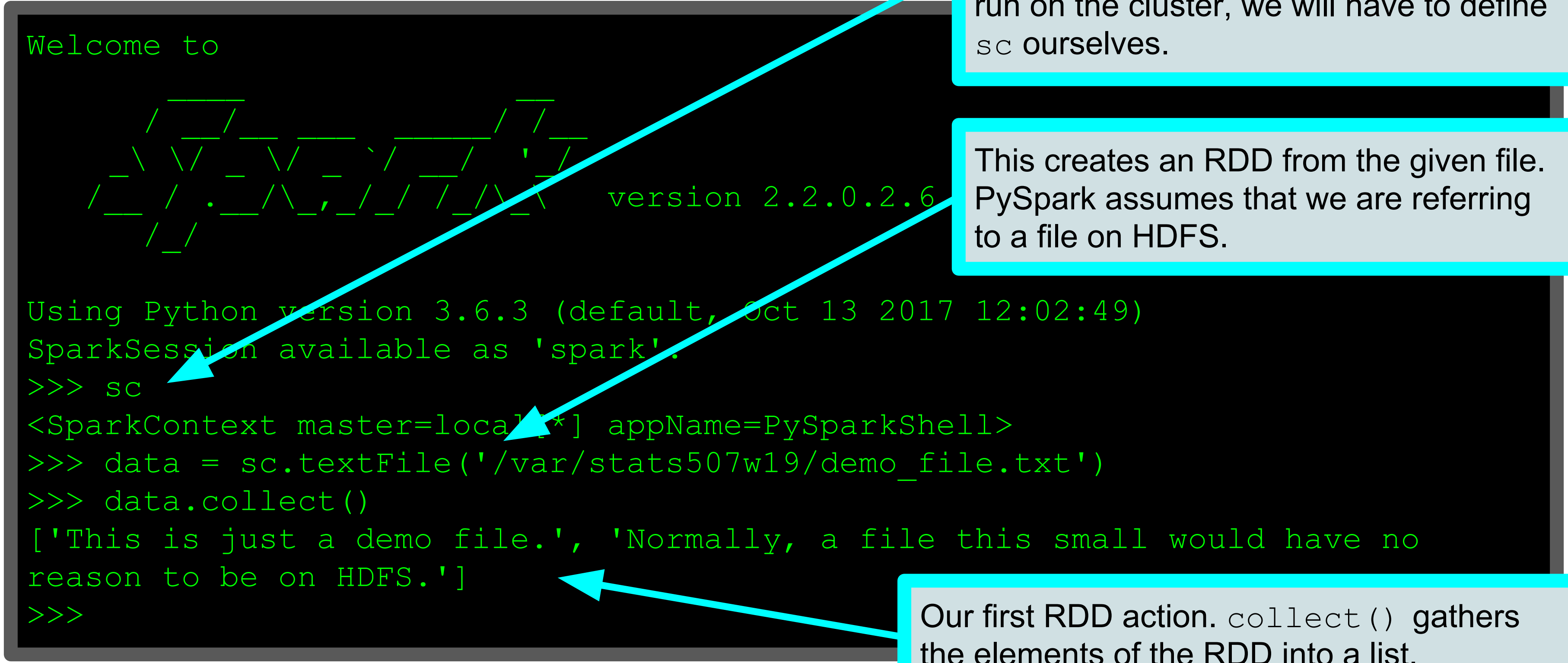
Spark finishes setting up our interactive session and gives us a prompt like the Python interpreter.

Creating an RDD from a file



```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0.2.6.3.0-235
      /_/

Using Python version 3.6.3 (default, Oct 13 2017 12:02:49)
SparkSession available as 'spark'.
>>> sc
<SparkContext master=local[*] appName=PySparkShell>
>>> data = sc.textFile('/var/stats507w19/demo_file.txt')
>>> data.collect()
['This is just a demo file.', 'Normally, a file this small would have no
reason to be on HDFS.']
>>>
```

Creating an RDD from a file



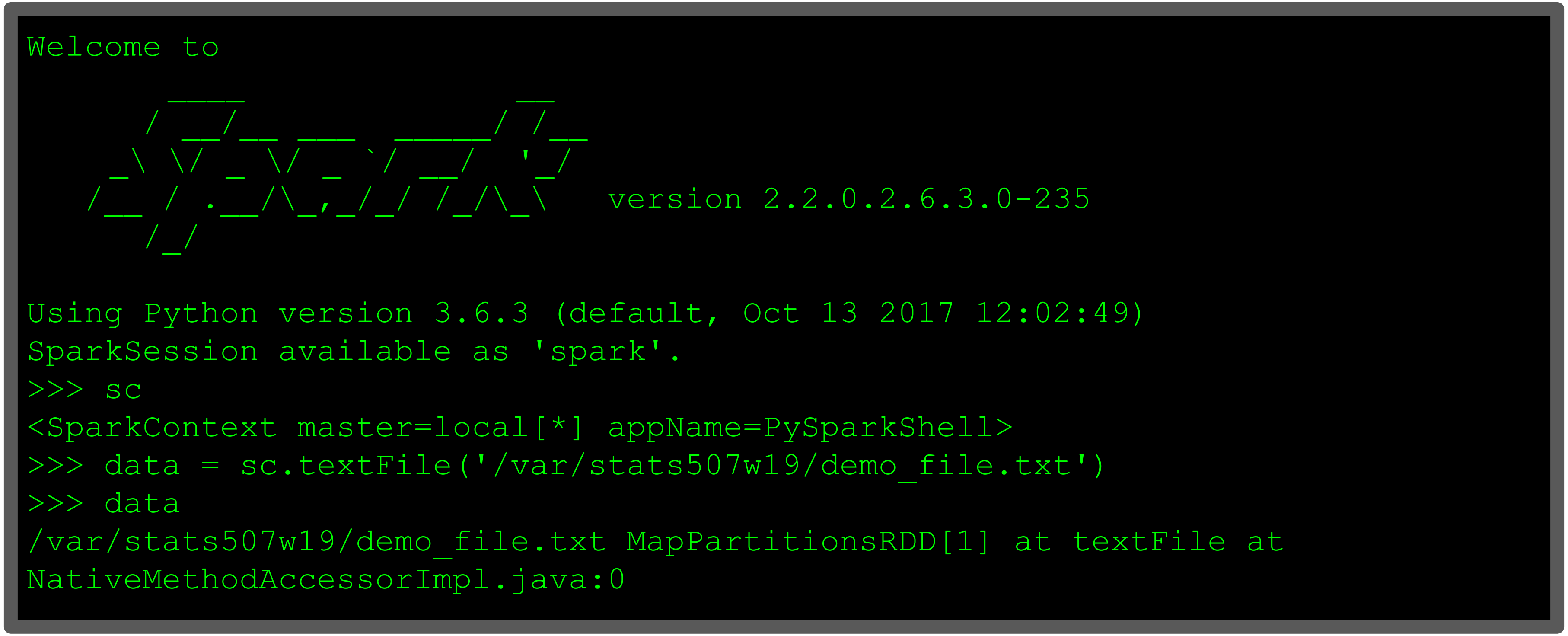
```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0.2.6
      /_/

Using Python version 3.6.3 (default, Oct 13 2017 12:02:49)
SparkSession available as 'spark'.
>>> sc
<SparkContext master=local[*] appName=PySparkShell>
>>> data = sc.textFile('/var/stats507w19/demo_file.txt')
>>> data.collect()
['This is just a demo file.', 'Normally, a file this small would have no
reason to be on HDFS.']
>>>
```

SparkContext is automatically created by the PySpark interpreter, and saved in the variable `sc`. When we write a job to be run on the cluster, we will have to define `sc` ourselves.

This creates an RDD from the given file. PySpark assumes that we are referring to a file on HDFS.

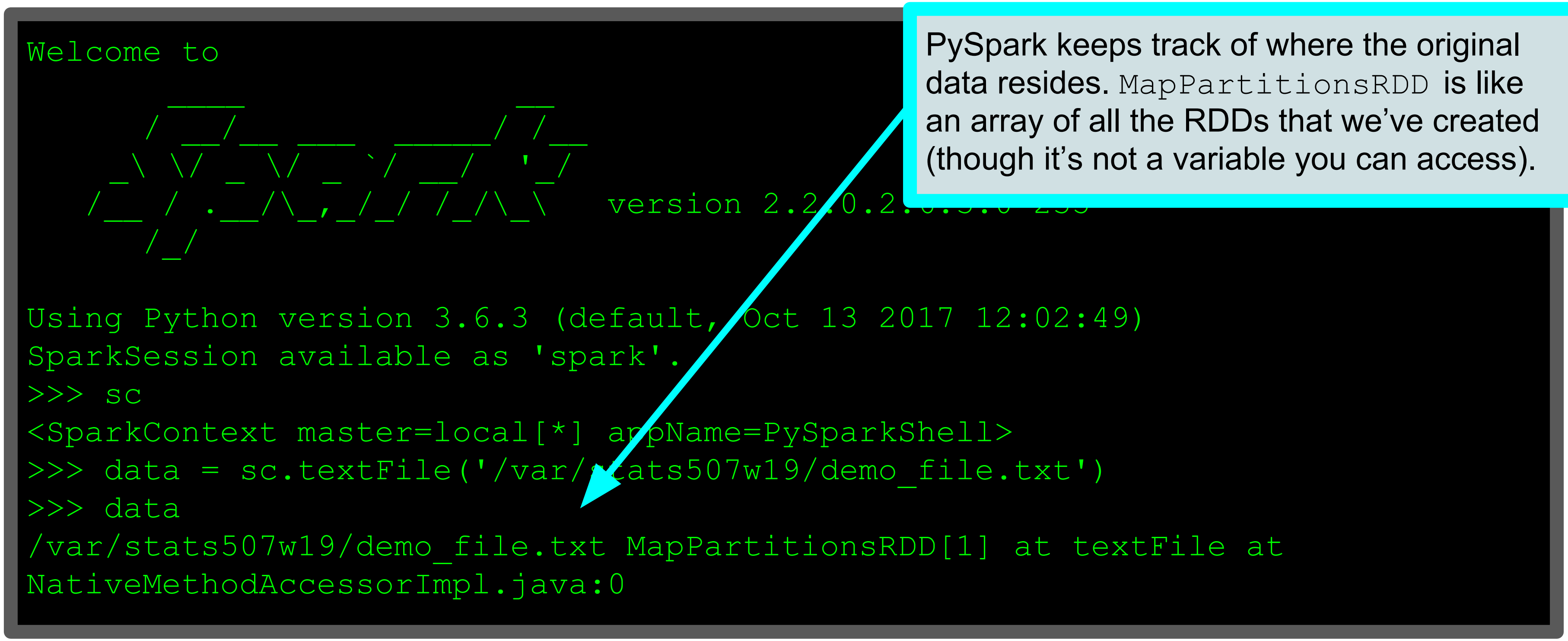
Our first RDD action. `collect()` gathers the elements of the RDD into a list.

PySpark keeps track of RDDs



```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0.2.6.3.0-235
      /_/

Using Python version 3.6.3 (default, Oct 13 2017 12:02:49)
SparkSession available as 'spark'.
>>> sc
<SparkContext master=local[*] appName=PySparkShell>
>>> data = sc.textFile('/var/stats507w19/demo_file.txt')
>>> data
/var/stats507w19/demo_file.txt MapPartitionsRDD[1] at textFile at
NativeMethodAccessorImpl.java:0
```

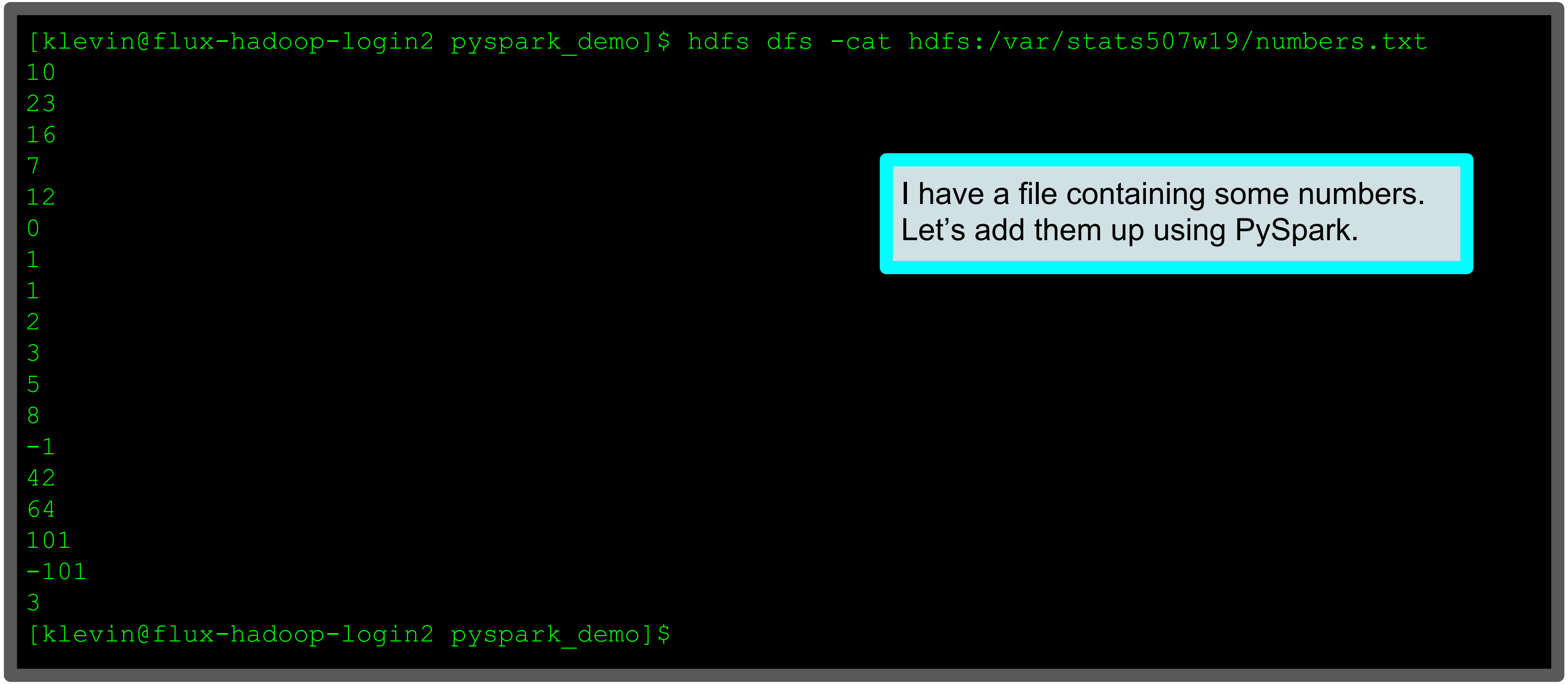
PySpark keeps track of RDDs



```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0.2.6.3.0-235
      /_/

Using Python version 3.6.3 (default, Oct 13 2017 12:02:49)
SparkSession available as 'spark'.
>>> sc
<SparkContext master=local[*] appName=PySparkShell>
>>> data = sc.textFile('/var/stats507w19/demo_file.txt')
>>> data
/var/stats507w19/demo_file.txt MapPartitionsRDD[1] at textFile at
NativeMethodAccessorImpl.java:0
```

PySpark keeps track of where the original data resides. `MapPartitionsRDD` is like an array of all the RDDs that we've created (though it's not a variable you can access).

Simple MapReduce task: Summations



```
[klevin@flux-hadoop-login2 pyspark_demo]$ hdfs dfs -cat hdfs:/var/stats507w19/numbers.txt
10
23
16
7
12
0
1
1
2
3
5
8
-1
42
64
101
-101
3
[klevin@flux-hadoop-login2 pyspark_demo]$
```

I have a file containing some numbers. Let’s add them up using PySpark.

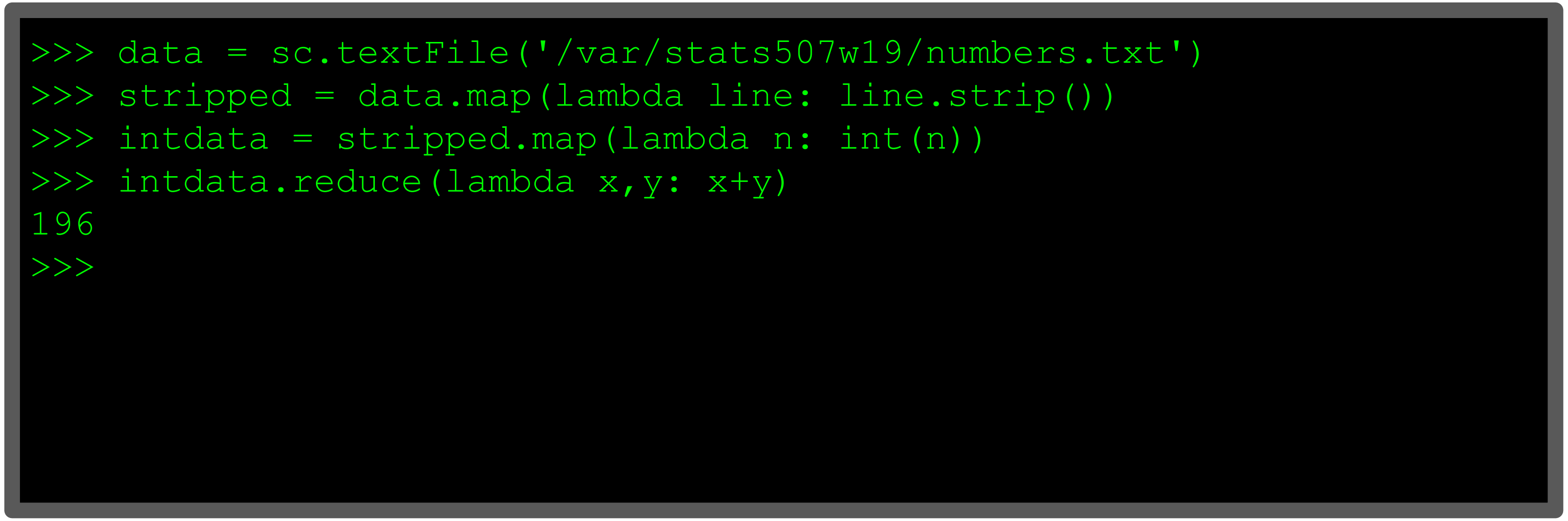
Simple MapReduce task: Summations

```
Using Python version 3.6.3 (default, Oct 13 2017 12:02:49)
SparkSession available as 'spark'.
>>> data = sc.textFile('/var/stats507w19/numbers.txt')
>>> data.collect()
['10', '23', '16', '7', '12', '0', '1', '1', '2', '3', '5', '8', '-1', '42', '64', '101',
'-101', '3']
>>> stripped = data.map(lambda line: line.strip())
>>> stripped.collect()
['10', '23', '16', '7', '12', '0', '1', '1', '2', '3', '5', '8', '-1', '42', '64', '101',
'-101', '3']
>>>
```

Using `strip()` here is redundant: PySpark automatically splits on whitespace when it reads from a text file. This is again just to show an example.

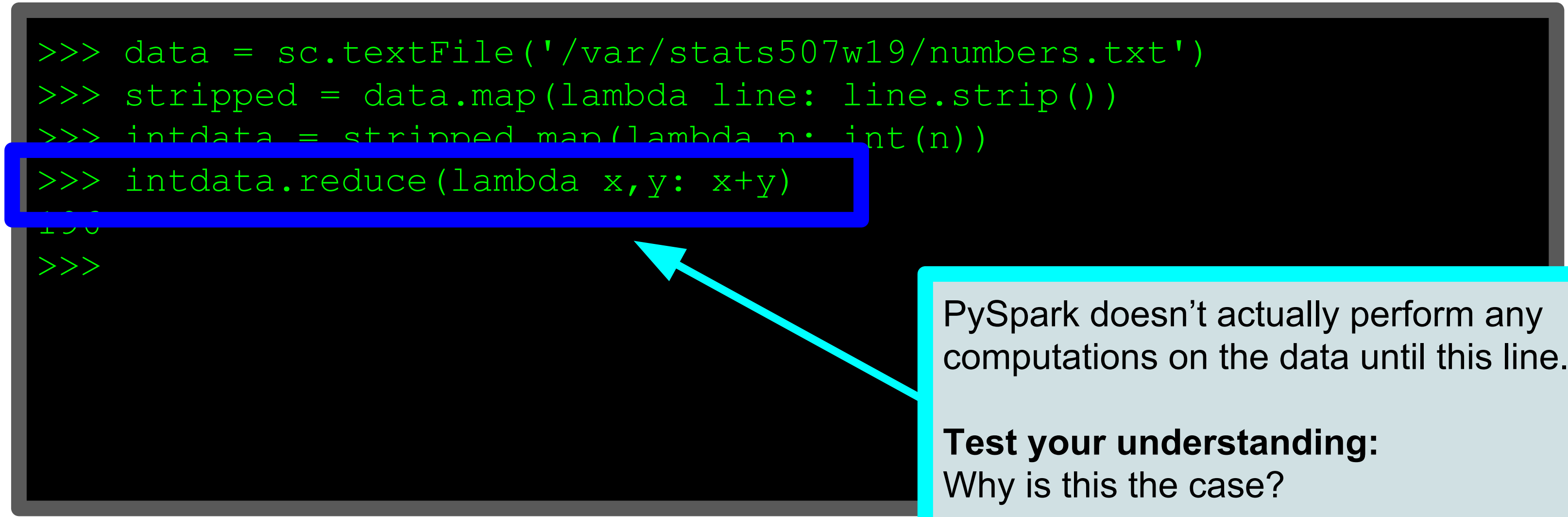
Reminder: `collect()` is an RDD action that produces a list of the RDD elements.

Simple MapReduce task: Summations



```
>>> data = sc.textFile('/var/stats507w19/numbers.txt')
>>> stripped = data.map(lambda line: line.strip())
>>> intdata = stripped.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
>>>
```

Simple MapReduce task: Summations

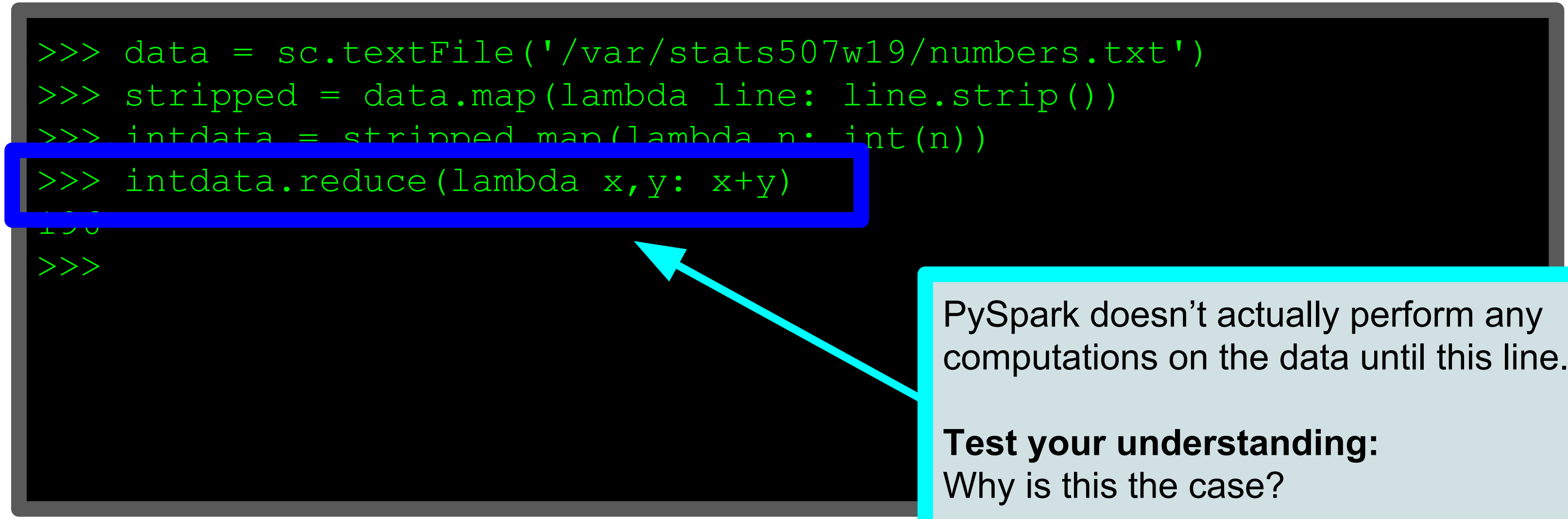


```
>>> data = sc.textFile('/var/stats507w19/numbers.txt')
>>> stripped = data.map(lambda line: line.strip())
>>> intdata = stripped.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
>>>
```

PySpark doesn’t actually perform any computations on the data until this line.

Test your understanding:
Why is this the case?

Simple MapReduce task: Summations



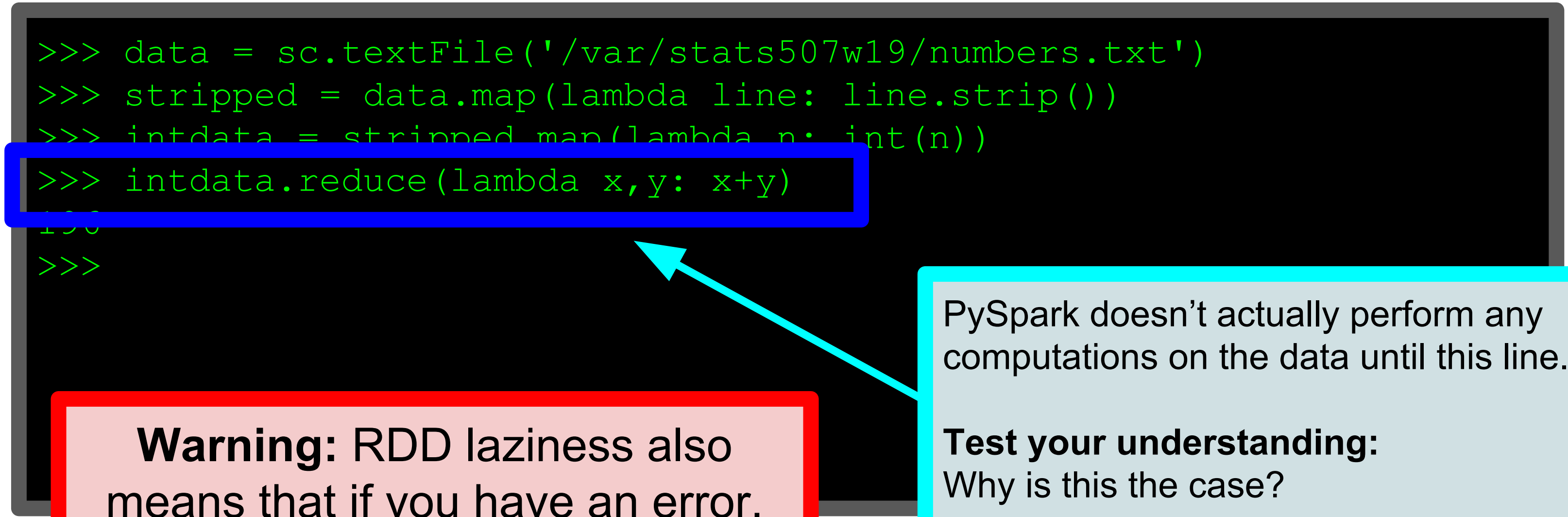
```
>>> data = sc.textFile('/var/stats507w19/numbers.txt')
>>> stripped = data.map(lambda line: line.strip())
>>> intdata = stripped.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
>>>
```

PySpark doesn’t actually perform any computations on the data until this line.

Test your understanding:
Why is this the case?

Answer: Because PySpark RDD operations are lazy, PySpark doesn’t perform any computations until we actually ask it for something via an **RDD action**.

Simple MapReduce task: Summations



```
>>> data = sc.textFile('/var/stats507w19/numbers.txt')
>>> stripped = data.map(lambda line: line.strip())
>>> intdata = stripped.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
>>>
```

PySpark doesn't actually perform any computations on the data until this line.

Test your understanding:
Why is this the case?

Warning: RDD laziness also means that if you have an error, you often won't find out about it until you call an RDD action!

Answer: Because PySpark RDD operations are lazy, PySpark doesn't perform any computations until we actually ask it for something via an **RDD action**.

Simple MapReduce task: Summations

```
>>> data = sc.textFile('/var/stats507w19/numbers.txt')
>>> stripped = data.map(lambda line: line.strip())
>>> intdata = stripped.map(lambda n: int(n))
>>> intdata.reduce(lambda x,y: x+y)
196
>>>
```

The Spark way of doing things also means that I can write all of the above much more succinctly.

```
>>> data = sc.textFile('/var/stats507w19/numbers.txt')
>>> data.map( lambda n: int(n)).reduce( lambda x,y:x+y )
196
```

Example RDD Transformations

`map`: apply a function to every element of the RDD

`filter`: retain only the elements satisfying a condition

`flatMap`: apply a map, but “flatten” the structure (details in a few slides)

`sample`: take a random sample from the elements of the RDD

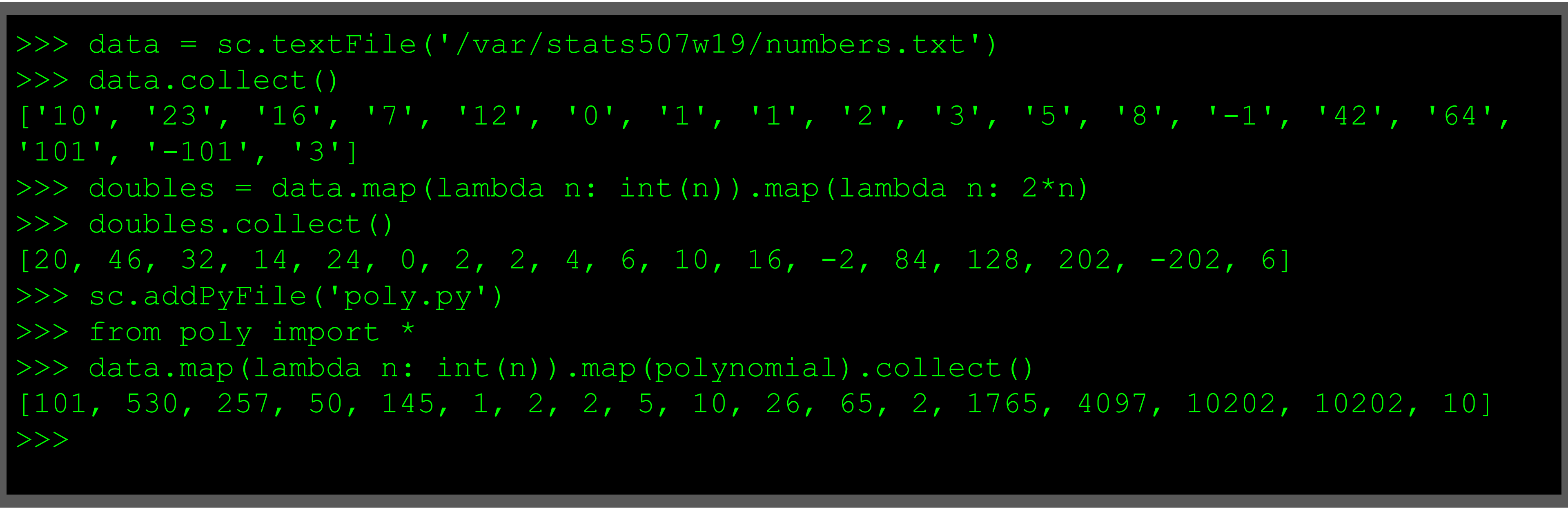
`distinct`: remove duplicate entries of the RDD

`reduceByKey`: on RDD of (K, V) pairs, return RDD of (K, V) pairs

values for each key are aggregated using the given reduce function.

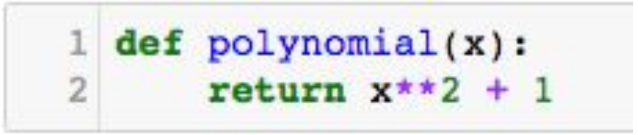
More: https://spark.apache.org/docs/0.9.0/scala-programming-guide.html#transformations

RDD.map()



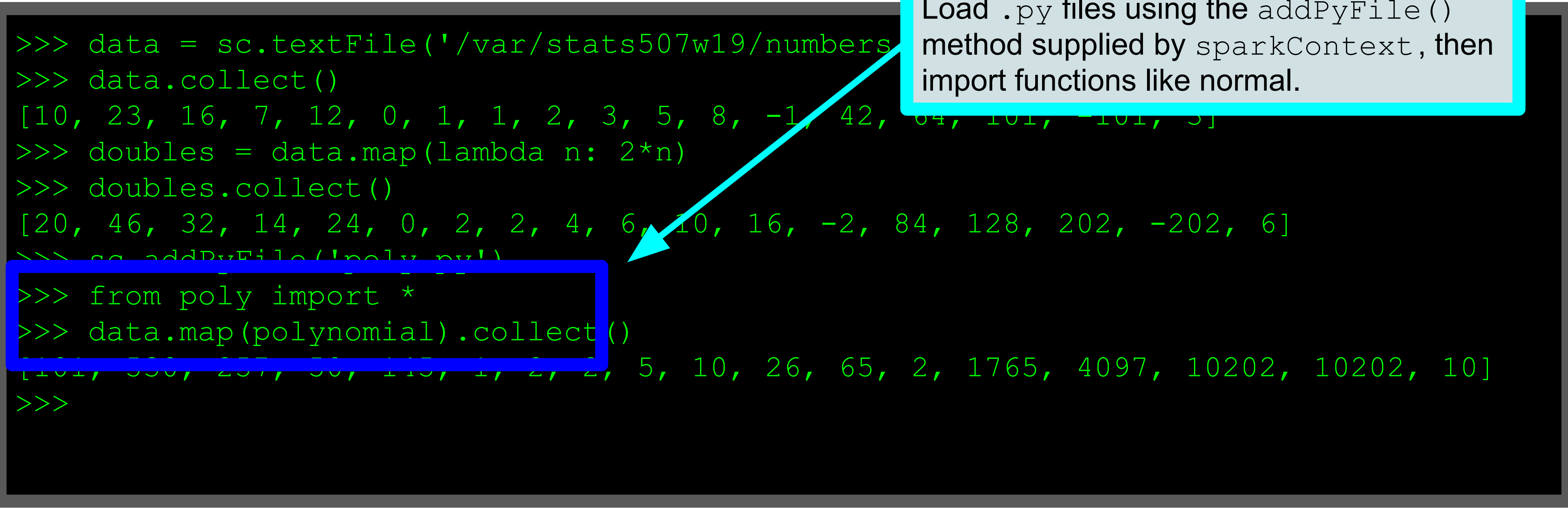
```
>>> data = sc.textFile('/var/stats507w19/numbers.txt')
>>> data.collect()
['10', '23', '16', '7', '12', '0', '1', '1', '2', '3', '5', '8', '-1', '42', '64',
'101', '-101', '3']
>>> doubles = data.map(lambda n: int(n)).map(lambda n: 2*n)
>>> doubles.collect()
[20, 46, 32, 14, 24, 0, 2, 2, 4, 6, 10, 16, -2, 84, 128, 202, -202, 6]
>>> sc.addPyFile('poly.py')
>>> from poly import *
>>> data.map(lambda n: int(n)).map(polynomial).collect()
[101, 530, 257, 50, 145, 1, 2, 2, 5, 10, 26, 65, 2, 1765, 4097, 10202, 10202, 10]
>>>
```

poly.py



```python
def polynomial(x):
    return x**2 + 1
```

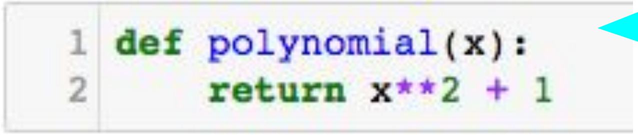
RDD.map()



```
>>> data = sc.textFile('/var/stats507w19/numbers
>>> data.collect()
[10, 23, 16, 7, 12, 0, 1, 1, 2, 3, 5, 8, -1, 42, 64, 101, -101, 3]
>>> doubles = data.map(lambda n: 2*n)
>>> doubles.collect()
[20, 46, 32, 14, 24, 0, 2, 2, 4, 6, 10, 16, -2, 84, 128, 202, -202, 6]
>>> sc.addPyFile('poly.py')
>>> from poly import *
>>> data.map(polynomial).collect()
[101, 530, 257, 50, 145, 1, 2, 2, 5, 10, 26, 65, 2, 1765, 4097, 10202, 10202, 10]
>>>
```

Load .py files using the addPyFile() method supplied by sparkContext, then import functions like normal.

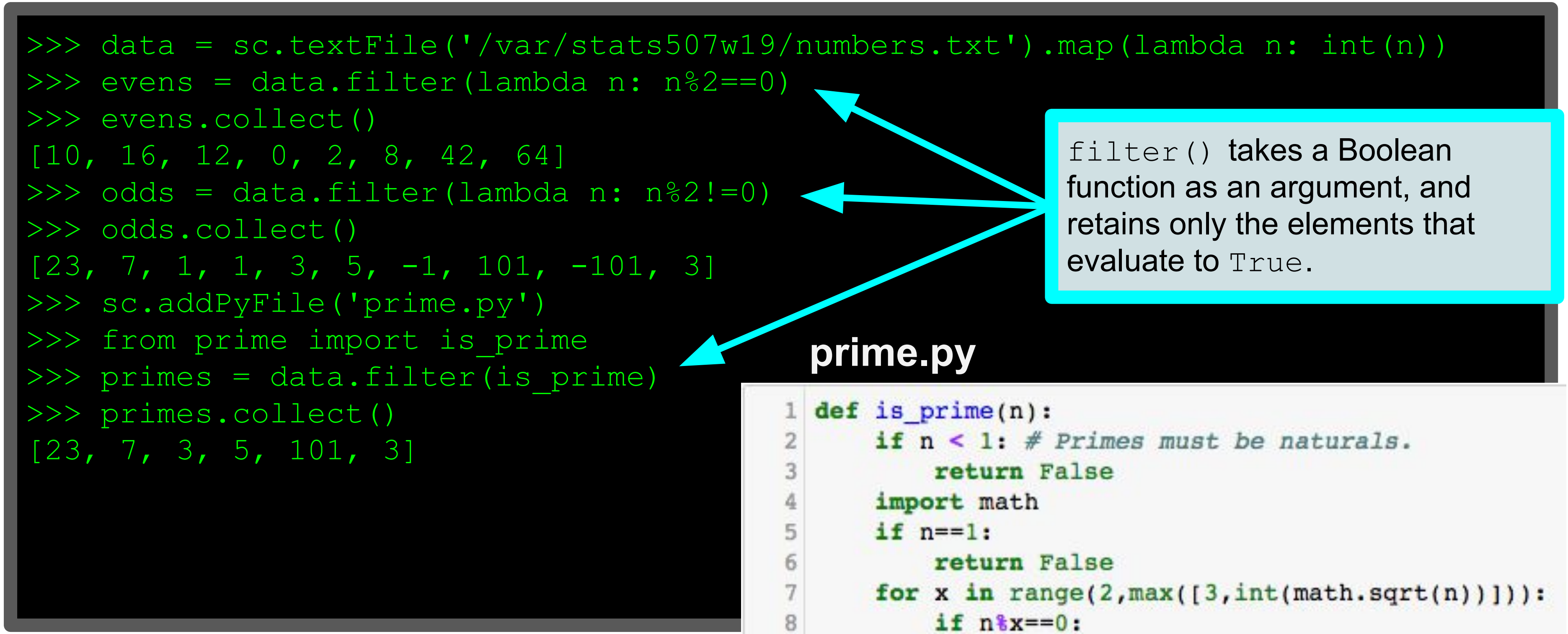
poly.py



```python
def polynomial(x):
    return x**2 + 1
```

This file is saved in the directory where I launched pyspark. If it's somewhere else, we have to specify the path to it.

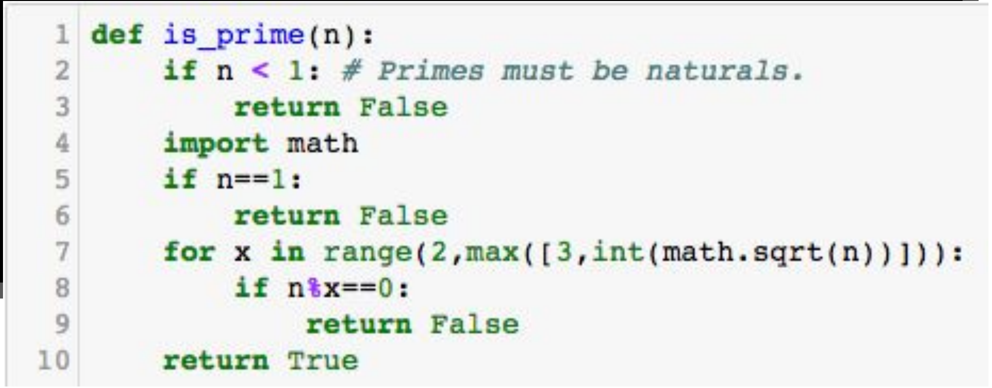
RDD.filter()



```
>>> data = sc.textFile('/var/stats507w19/numbers.txt').map(lambda n: int(n))
>>> evens = data.filter(lambda n: n%2==0)
>>> evens.collect()
[10, 16, 12, 0, 2, 8, 42, 64]
>>> odds = data.filter(lambda n: n%2!=0)
>>> odds.collect()
[23, 7, 1, 1, 3, 5, -1, 101, -101, 3]
>>> sc.addPyFile('prime.py')
>>> from prime import is_prime
>>> primes = data.filter(is_prime)
>>> primes.collect()
[23, 7, 3, 5, 101, 3]
```

`filter()` takes a Boolean function as an argument, and retains only the elements that evaluate to `True`.

prime.py



```python
def is_prime(n):
    if n < 1: # Primes must be naturals.
        return False
    import math
    if n==1:
        return False
    for x in range(2,max([3,int(math.sqrt(n))])):
        if n%x==0:
            return False
    return True
```

RDD.sample()

```
>>> data = sc.textFile('/var/stats507w19/numbers.txt').map(lambda n: int(n))
>>> samp = data.sample(False, 0.5)
>>> samp.collect()
[12, 5, -1, 42, 101, -101]
>>> samp = data.sample(True, 0.5)
>>> samp.collect()
[10, 10, 23, 7, 2, 42, 101, 3]
>>>
```

`sample(`*`withReplacement`*`, `*`fraction`*`, [`*`seed`*`])`

`RDD.sample()` is mostly useful for testing on small subsets of your data.

Dealing with more complicated elements

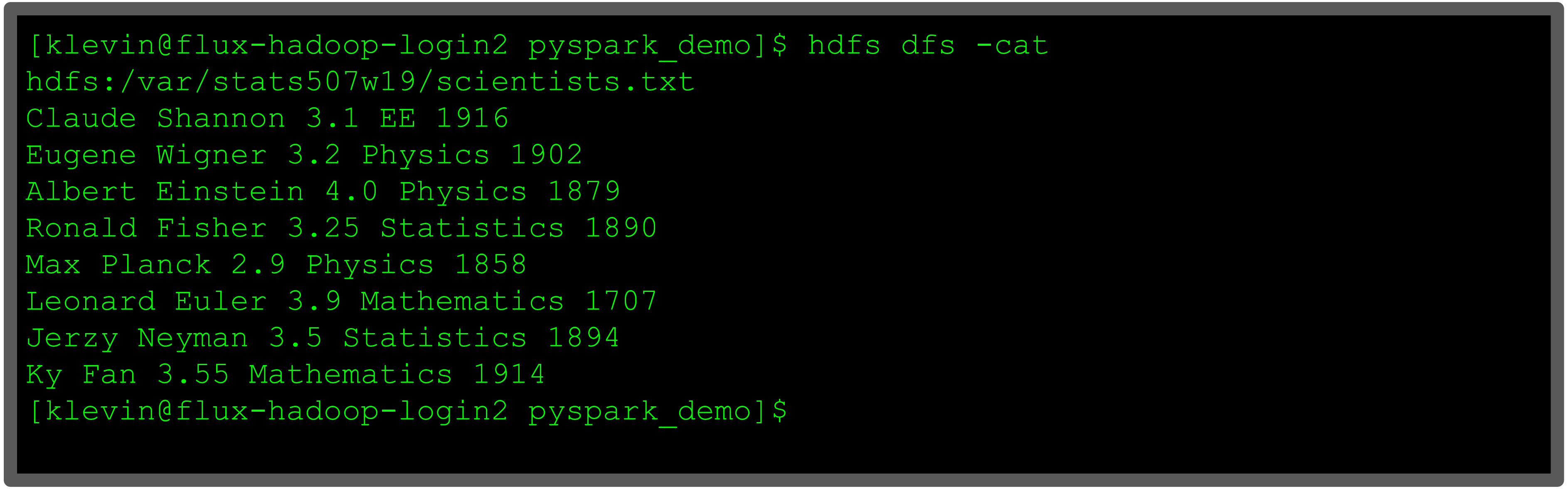
What if the elements of my RDD are more complicated than just numbers?...

> **Example:** if I have a comma-separated database-like file

Short answer: RDD elements are always tuples

But what about *really* complicated elements?
Recall that PySpark RDDs are immutable. This means that if you want your RDD to contain, for example, python dictionaries, you need to do a bit of extra work to turn Python objects into strings via **serialization**, which you already know about from the `pickle` module:
https://docs.python.org/3/library/pickle.html

Database-like file

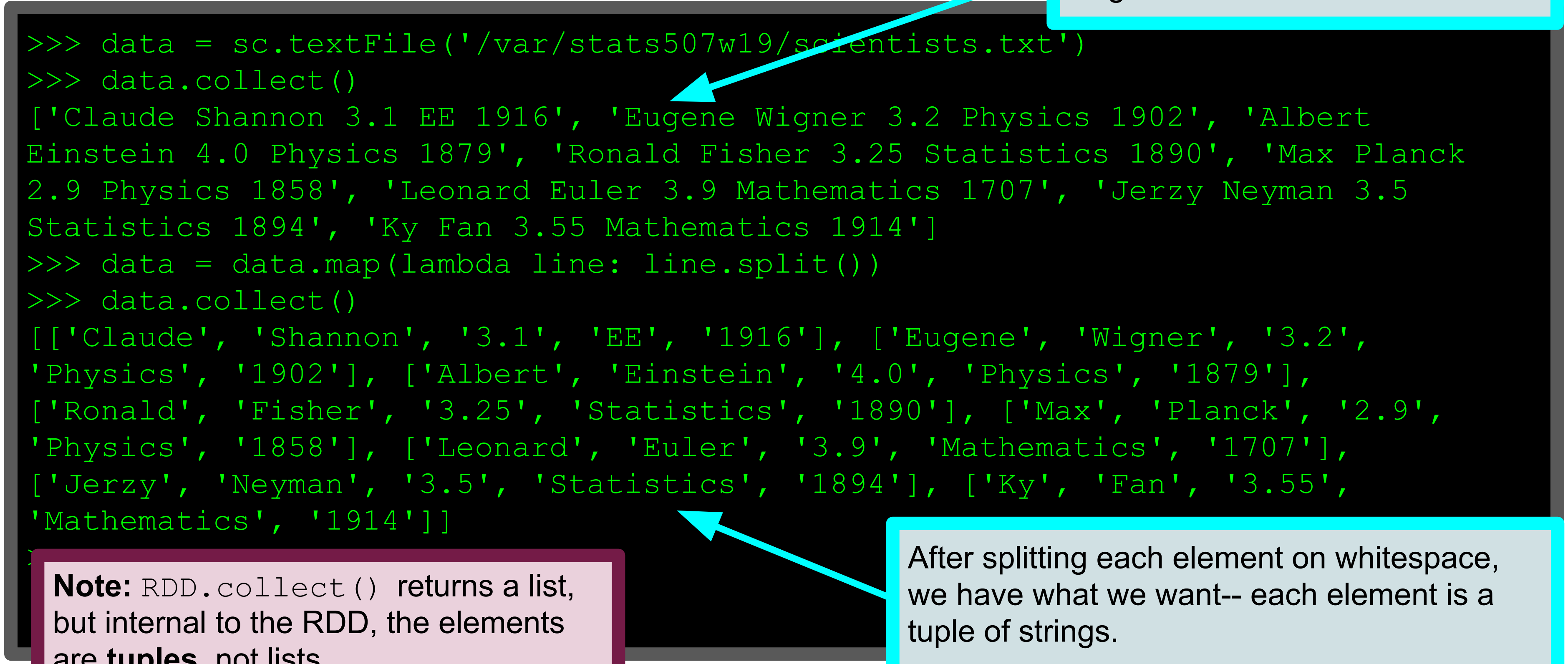


```
[klevin@flux-hadoop-login2 pyspark_demo]$ hdfs dfs -cat
hdfs:/var/stats507w19/scientists.txt
Claude Shannon 3.1 EE 1916
Eugene Wigner 3.2 Physics 1902
Albert Einstein 4.0 Physics 1879
Ronald Fisher 3.25 Statistics 1890
Max Planck 2.9 Physics 1858
Leonard Euler 3.9 Mathematics 1707
Jerzy Neyman 3.5 Statistics 1894
Ky Fan 3.55 Mathematics 1914
[klevin@flux-hadoop-login2 pyspark_demo]$
```

Database-like file

```
>>> data = sc.textFile('/var/stats507w19/scientists.txt')
>>> data.collect()
['Claude Shannon 3.1 EE 1916', 'Eugene Wigner 3.2 Physics 1902', 'Albert
Einstein 4.0 Physics 1879', 'Ronald Fisher 3.25 Statistics 1890', 'Max Planck
2.9 Physics 1858', 'Leonard Euler 3.9 Mathematics 1707', 'Jerzy Neyman 3.5
Statistics 1894', 'Ky Fan 3.55 Mathematics 1914']
>>> data = data.map(lambda line: line.split())
>>> data.collect()
[['Claude', 'Shannon', '3.1', 'EE', '1916'], ['Eugene', 'Wigner', '3.2',
'Physics', '1902'], ['Albert', 'Einstein', '4.0', 'Physics', '1879'],
['Ronald', 'Fisher', '3.25', 'Statistics', '1890'], ['Max', 'Planck', '2.9',
'Physics', '1858'], ['Leonard', 'Euler', '3.9', 'Mathematics', '1707'],
['Jerzy', 'Neyman', '3.5', 'Statistics', '1894'], ['Ky', 'Fan', '3.55',
'Mathematics', '1914']]
>>>
```

Database-like file



```
>>> data = sc.textFile('/var/stats507w19/scientists.txt')
>>> data.collect()
['Claude Shannon 3.1 EE 1916', 'Eugene Wigner 3.2 Physics 1902', 'Albert
Einstein 4.0 Physics 1879', 'Ronald Fisher 3.25 Statistics 1890', 'Max Planck
2.9 Physics 1858', 'Leonard Euler 3.9 Mathematics 1707', 'Jerzy Neyman 3.5
Statistics 1894', 'Ky Fan 3.55 Mathematics 1914']
>>> data = data.map(lambda line: line.split())
>>> data.collect()
[['Claude', 'Shannon', '3.1', 'EE', '1916'], ['Eugene', 'Wigner', '3.2',
'Physics', '1902'], ['Albert', 'Einstein', '4.0', 'Physics', '1879'],
['Ronald', 'Fisher', '3.25', 'Statistics', '1890'], ['Max', 'Planck', '2.9',
'Physics', '1858'], ['Leonard', 'Euler', '3.9', 'Mathematics', '1707'],
['Jerzy', 'Neyman', '3.5', 'Statistics', '1894'], ['Ky', 'Fan', '3.55',
'Mathematics', '1914']]
>
```

On initial read, each line is a single element in the RDD.

After splitting each element on whitespace, we have what we want-- each element is a tuple of strings.

Note: `RDD.collect()` returns a list, but internal to the RDD, the elements are **tuples**, not lists.

RDD.distinct()

```
>>> data = sc.textFile('/var/stats507w19/scientists.txt')
>>> data = data.map(lambda line: line.split())
>>> fields = data.map(lambda t: t[3]).distinct()
>>> fields.collect()
['EE', 'Statistics', 'Physics', 'Mathematics']
```

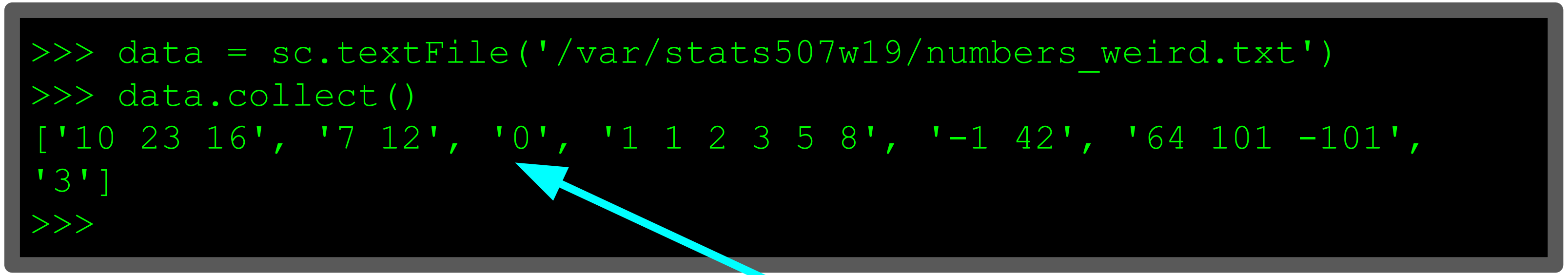
RDD.distinct()

Each tuple is of the form
(first_name, last_name, GPA, field, birth_year)

```
>>> data = sc.textFile('/var/stats507/w19/scientists.txt')
>>> data = data.map(lambda line: line.split())
>>> fields = data.map(lambda t: t[3]).distinct()
>>> fields.collect()
['EE', 'Statistics', 'Physics', 'Mathematics']
```

`RDD.distinct()` does just what you think it does!

`RDD.flatMap()`



```
>>> data = sc.textFile('/var/stats507w19/numbers_weird.txt')
>>> data.collect()
['10 23 16', '7 12', '0', '1 1 2 3 5 8', '-1 42', '64 101 -101',
'3']
>>>
```

Same list of numbers, but they’re not one per line, anymore...

From PySpark documentation:
flatMap(*func*) Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

RDD.flatMap()

```
>>> data = sc.textFile('/var/stats507w19/numbers_weird.txt')
>>> data.collect()
['10 23 16', '7 12', '0', '1 1 2 3 5 8', '-1 42', '64 101 -101', '3']
>>> flattened = data.flatMap(lambda line: [x for x in line.split()])
>>> flattened.collect()
['10', '23', '16', '7', '12', '0', '1', '1', '2', '3', '5', '8', '-1',
'42', '64', '101', '-101', '3']
>>> flattened.map(lambda n: int(n)).reduce(lambda x,y: x+y)
196
>>>
```

So we can think of `flatMap()` as producing a list for each element in the RDD, and then concatenating those lists. But crucially, the output is another RDD, **not** a list. This kind of operation is called **flattening**, and it's a common pattern in functional programming.

Example RDD Actions

`reduce`: aggregate elements of the RDD using a function

`collect`: return all elements of the RDD as an array at the driver program.

`count`: return the number of elements in the RDD.

`countByKey`: Returns <key, int> pairs with count of each key.

Only available on RDDs with elements of the form <key,value>

More: https://spark.apache.org/docs/0.9.0/scala-programming-guide.html#actions

RDD.count()

```
>>> data = sc.textFile('/var/stats507w19/demo_file.txt')
>>> data = data.flatMap(lambda line: line.split())
>>> data = data.map(lambda w: w.lower())
>>> data.collect()
['this', 'is', 'just', 'a', 'demo', 'file.', 'normally,', 'a',
'file', 'this', 'small', 'would', 'have', 'no', 'reason', 'to',
'be', 'on', 'hdfs.']
>>> uniqwords = data.distinct()
>>> uniqwords.count()
17
>>>
```

RDD.countByKey()

```
>>> data = sc.textFile('/var/stats507w19/demo_file.txt')
>>> data = data.flatMap(lambda line: line.split())
>>> data = data.map(lambda w: (w.lower(), 0))
>>> data.countByKey()
defaultdict(<class 'int'>, {'this': 2, 'is': 1, 'just': 1, 'a': 2,
'demo': 1, 'file.': 1, 'normally,': 1, 'file': 1, 'small': 1,
'would': 1, 'have': 1, 'no': 1, 'reason': 1, 'to': 1, 'be': 1,
'on': 1, 'hdfs.': 1})
>>>
```

Note: In the example above, each word had a key 0, but note that in the dictionary produced by `countByKey`, the values correspond to how many times that key appeared. This is because `countByKey()` counts how many times each key appears and **ignores their values**.

Running PySpark on the Cluster

So far, we’ve just been running in interactive mode.

Problem: Interactive mode is good for prototyping and testing…

...but not so well-suited for running large jobs.

Solution: PySpark can also be submitted to the grid and run there.

Instead of `pyspark`, we use `spark-submit` on the Fladoop grid.

Instructions specific to Fladoop can be found here:

http://arc-ts.umich.edu/hadoop-user-guide/#document-5

Two preliminaries

Before we can talk about running jobs on the cluster...

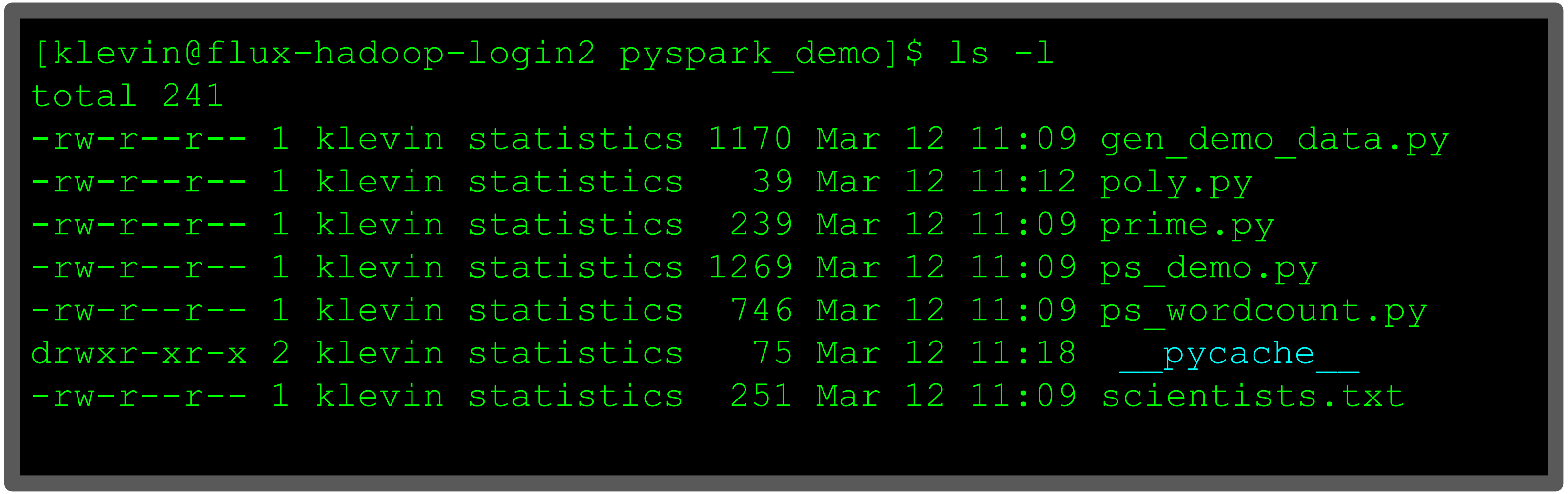
1) **UNIX groups**
   How we control who can and can't access files

2) **Queues on compute clusters**
   How we know who has to pay for compute time

UNIX Groups

On UNIX-like systems, files are owned by users

On UNIX/Linux/MacOS:



```
[klevin@flux-hadoop-login2 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

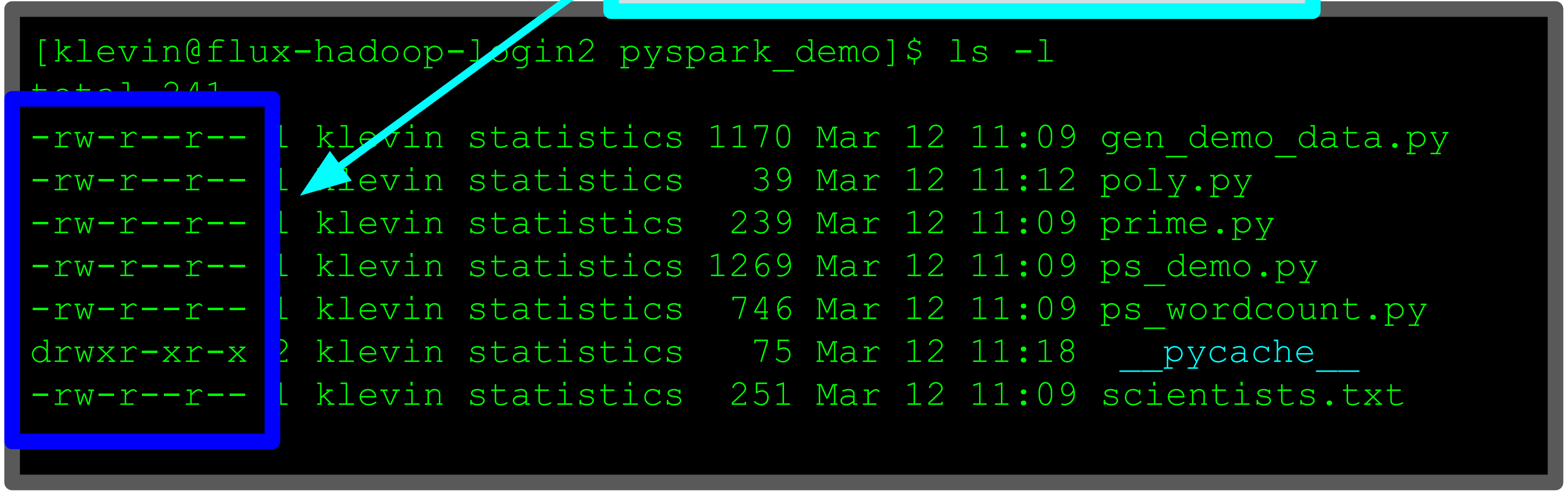
UNIX Groups

Legend
`d` : directory
`r` : read access
`w` : write access
`x` : execute access

On UNIX-like systems, files are owned by users

On UNIX/Linux/MacOS:

These lines are permission information.



```
[klevin@flux-hadoop-login2 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

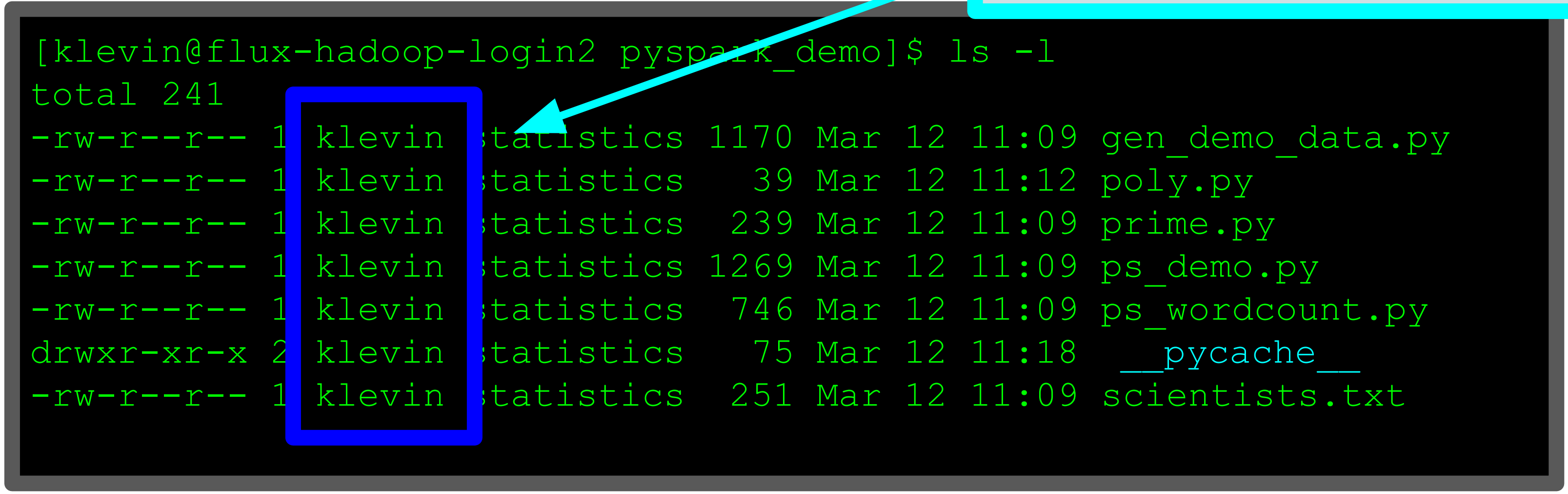
UNIX Groups

Legend
`d` : directory
`r` : read access
`w` : write access
`x` : execute access

On UNIX-like systems, files are owned by users

On UNIX/Linux/MacOS:

This column lists which user owns the file



```
[klevin@flux-hadoop-login2 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

UNIX Groups

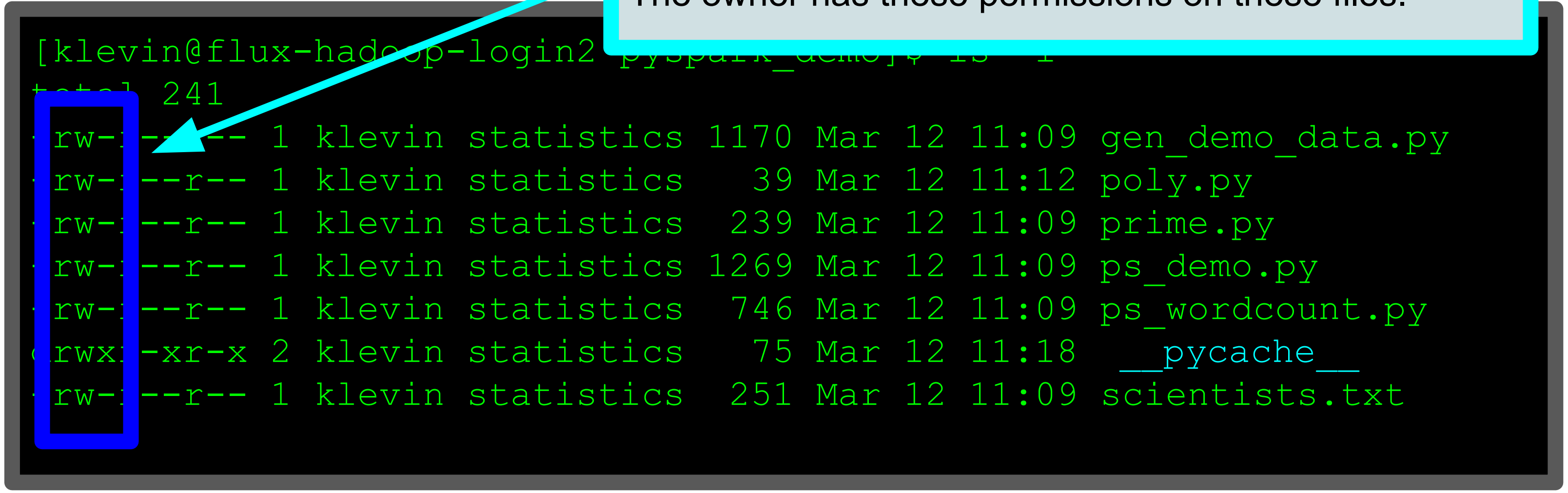
On UNIX-like systems, files are owned by users

Legend

`d`	: directory
`r`	: read access
`w`	: write access
`x`	: execute access

On UNIX/Linux/MacOS:

These specific columns specify owner permissions. The owner has these permissions on these files.



```
[klevin@flux-hadoop-login2 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

UNIX Groups

Legend

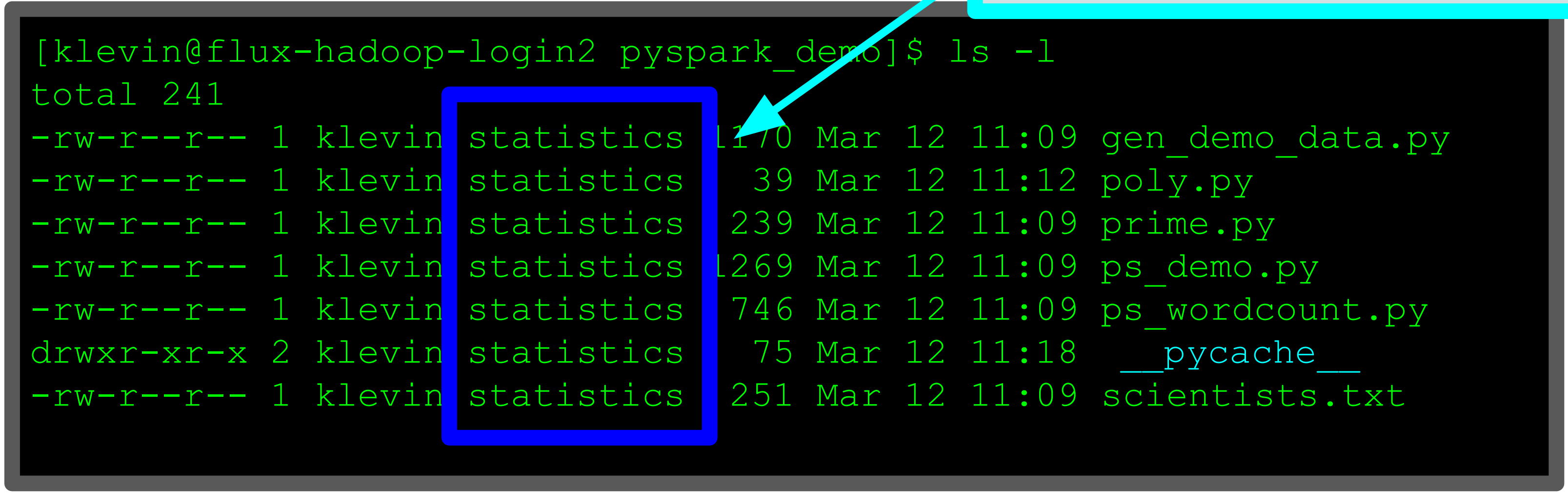
`d`	: directory
`r`	: read access
`w`	: write access
`x`	: execute access

On UNIX-like systems, files are owned by users

Sets of users, called **groups**, can be granted special permissions

On UNIX/Linux/MacOS:

This column lists what group owns the file



```
[klevin@flux-hadoop-login2 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

UNIX Groups

Legend

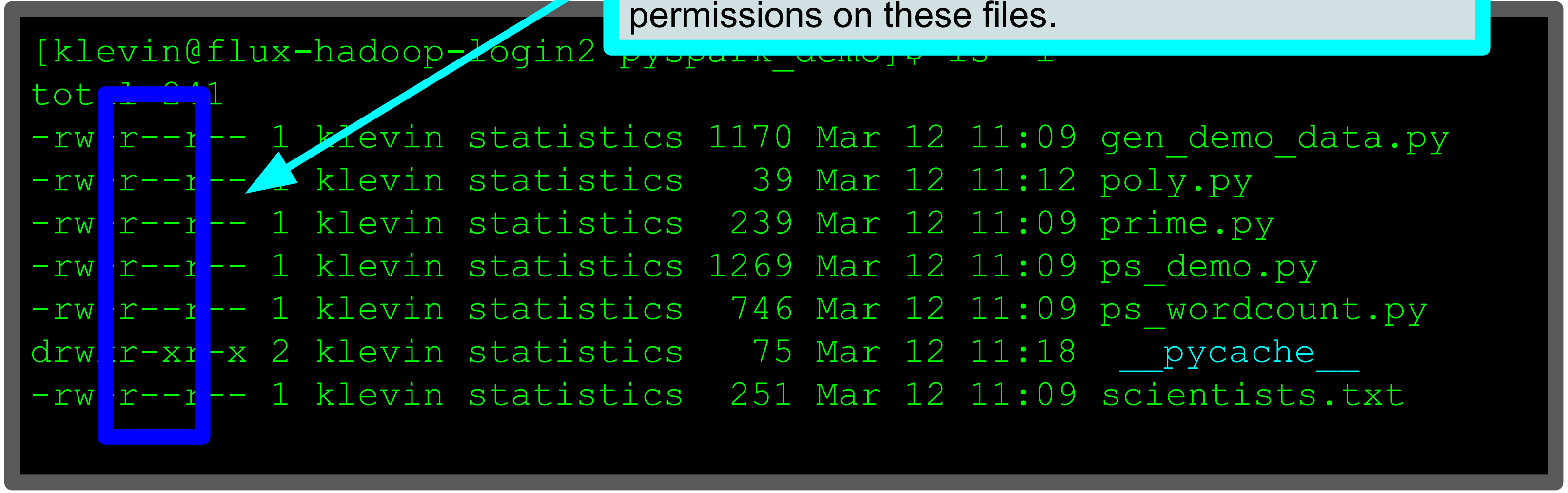
`d`	: directory
`r`	: read access
`w`	: write access
`x`	: execute access

On UNIX-like systems, files are owned by users

Sets of users, called **groups**, can be granted special permissions

On UNIX/Linux/MacOS:

These specific columns specify group permissions. Anyone in the `statistics` group has these permissions on these files.



```
[klevin@flux-hadoop-login2 pyspark_demo]$ ls -l
total 241
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

UNIX Groups

Legend

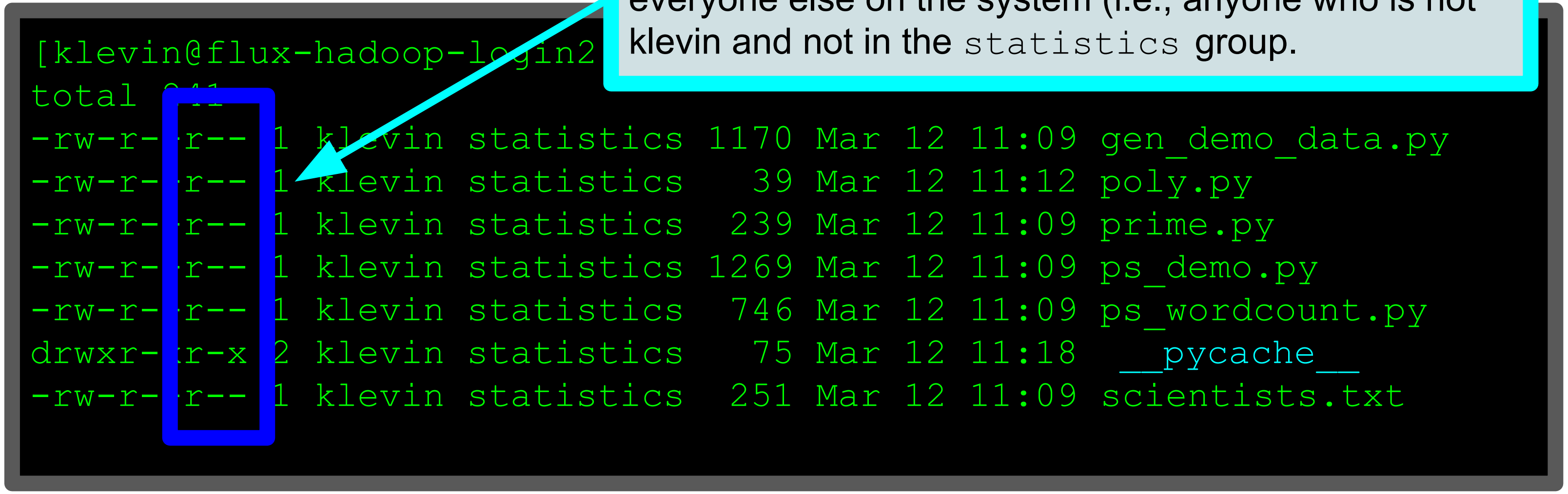
`d`	: directory
`r`	: read access
`w`	: write access
`x`	: execute access

On UNIX-like systems, files are owned by users

Sets of users, called **groups**, can be granted special permissions

On UNIX/Linux/MacOS:

These specific columns specify the permissions for everyone else on the system (i.e., anyone who is not klevin and not in the `statistics` group.



```
[klevin@flux-hadoop-login2
total 941
-rw-r--r-- 1 klevin statistics 1170 Mar 12 11:09 gen_demo_data.py
-rw-r--r-- 1 klevin statistics   39 Mar 12 11:12 poly.py
-rw-r--r-- 1 klevin statistics  239 Mar 12 11:09 prime.py
-rw-r--r-- 1 klevin statistics 1269 Mar 12 11:09 ps_demo.py
-rw-r--r-- 1 klevin statistics  746 Mar 12 11:09 ps_wordcount.py
drwxr-xr-x 2 klevin statistics   75 Mar 12 11:18 __pycache__
-rw-r--r-- 1 klevin statistics  251 Mar 12 11:09 scientists.txt
```

Cluster computing: queues

Compute cluster is a shared resource

How do we know who has to pay for what?

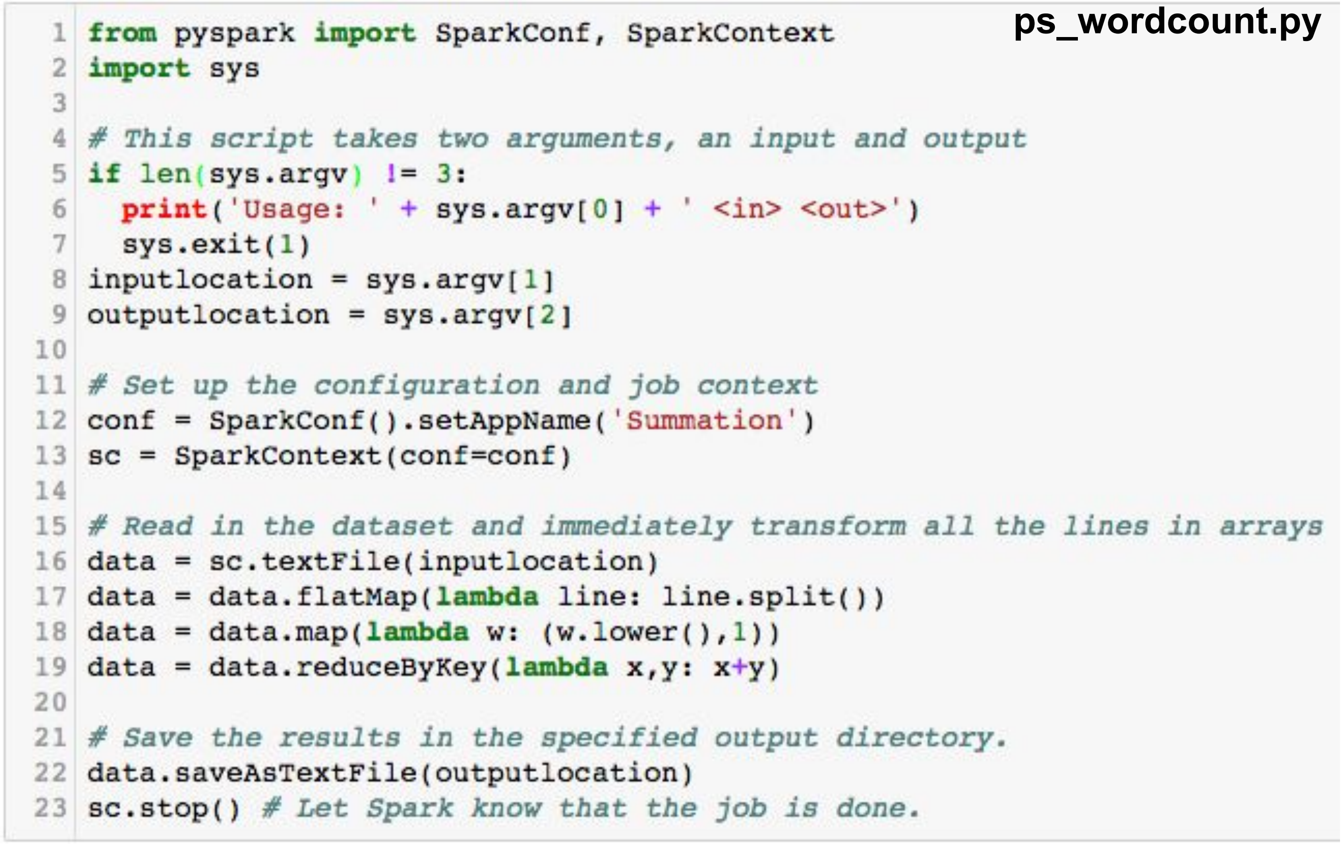
Flux operates what are called **allocations**, which are like pre-paid accounts

When you submit a job, you submit to a **queue**

- Like a line that you stand in to wait for your job to be run
- One line for each class, lab, etc

This semester, we are using the default queue.

Submitting to the queue: `spark-submit`

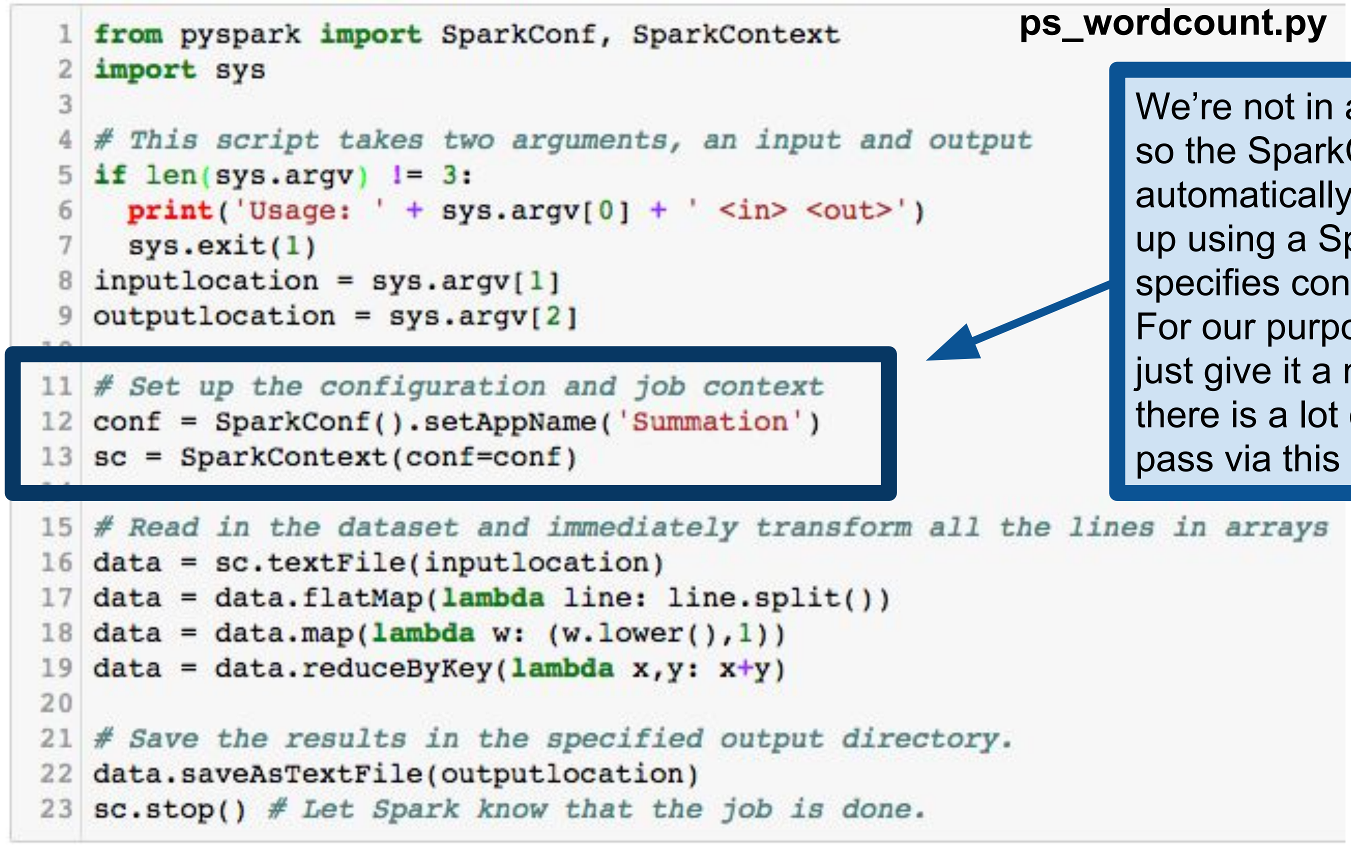


ps_wordcount.py

```python
from pyspark import SparkConf, SparkContext
import sys

# This script takes two arguments, an input and output
if len(sys.argv) != 3:
  print('Usage: ' + sys.argv[0] + ' <in> <out>')
  sys.exit(1)
inputlocation = sys.argv[1]
outputlocation = sys.argv[2]

# Set up the configuration and job context
conf = SparkConf().setAppName('Summation')
sc = SparkContext(conf=conf)

# Read in the dataset and immediately transform all the lines in arrays
data = sc.textFile(inputlocation)
data = data.flatMap(lambda line: line.split())
data = data.map(lambda w: (w.lower(),1))
data = data.reduceByKey(lambda x,y: x+y)

# Save the results in the specified output directory.
data.saveAsTextFile(outputlocation)
sc.stop() # Let Spark know that the job is done.
```

Submitting to the queue: `spark-submit`

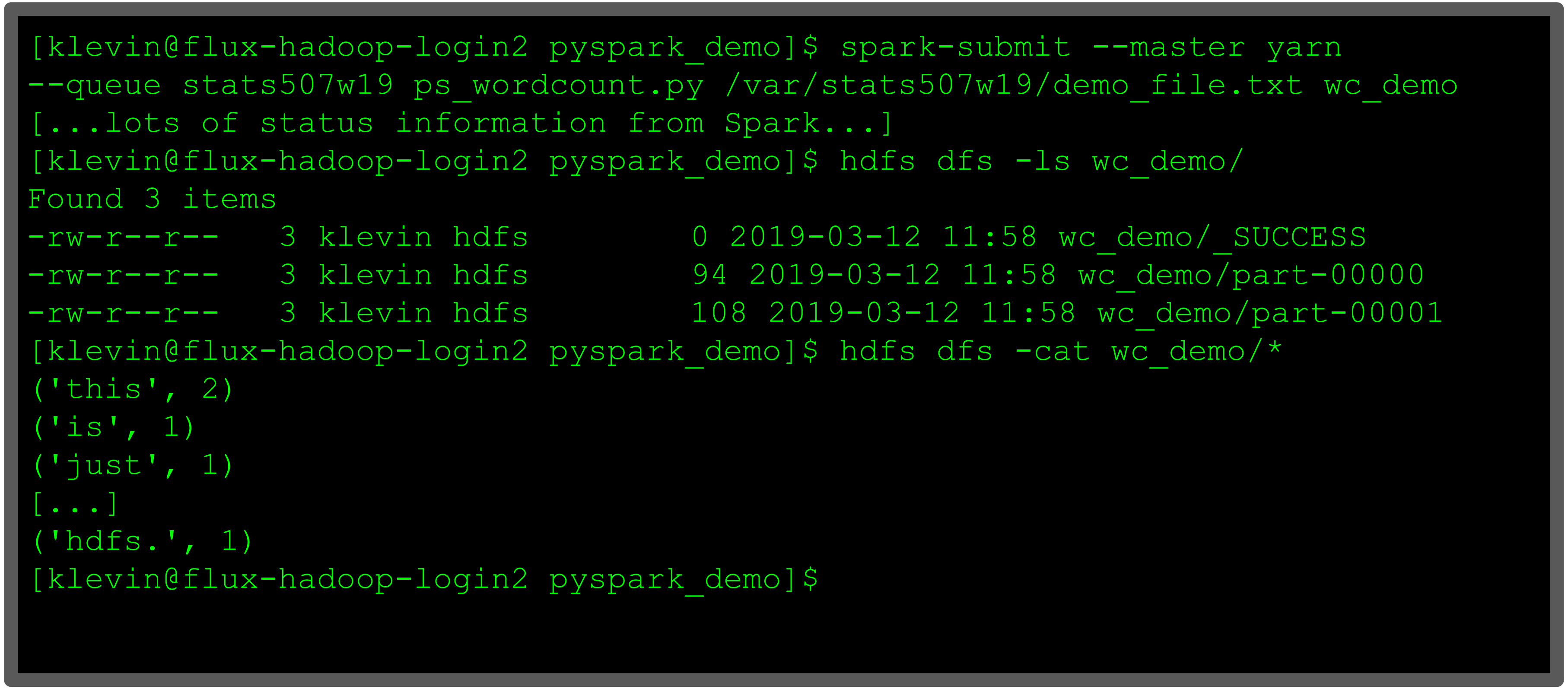
ps_wordcount.py



```python
from pyspark import SparkConf, SparkContext
import sys

# This script takes two arguments, an input and output
if len(sys.argv) != 3:
  print('Usage: ' + sys.argv[0] + ' <in> <out>')
  sys.exit(1)
inputlocation = sys.argv[1]
outputlocation = sys.argv[2]

# Set up the configuration and job context
conf = SparkConf().setAppName('Summation')
sc = SparkContext(conf=conf)

# Read in the dataset and immediately transform all the lines in arrays
data = sc.textFile(inputlocation)
data = data.flatMap(lambda line: line.split())
data = data.map(lambda w: (w.lower(),1))
data = data.reduceByKey(lambda x,y: x+y)

# Save the results in the specified output directory.
data.saveAsTextFile(outputlocation)
sc.stop() # Let Spark know that the job is done.
```

We're not in an interactive session, so the SparkContext isn't set up automatically. SparkContext is set up using a SparkConf object, which specifies configuration information. For our purposes, it's enough to just give it a name, but in general there is a lot of information we can pass via this object.

Submitting to the queue: `spark-submit`



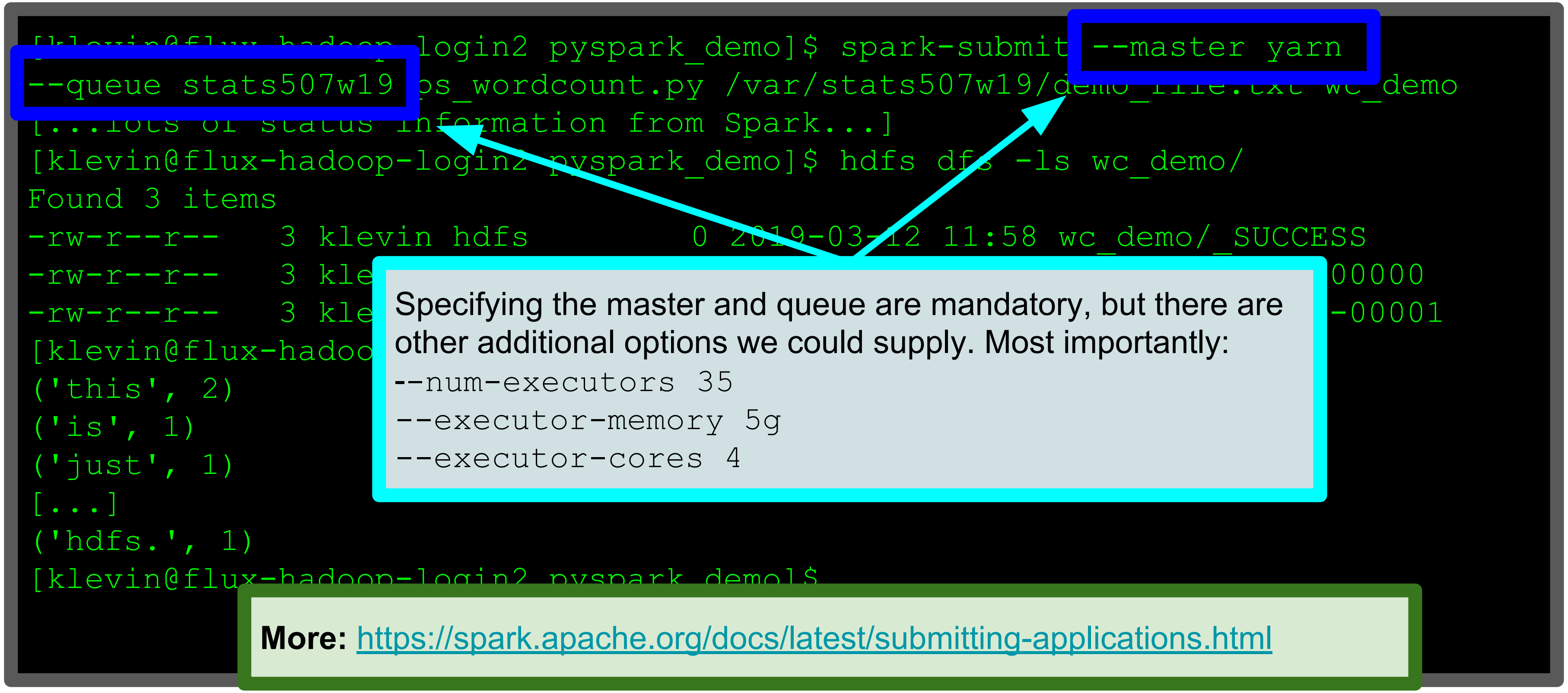
```
[klevin@flux-hadoop-login2 pyspark_demo]$ spark-submit --master yarn
--queue stats507w19 ps_wordcount.py /var/stats507w19/demo_file.txt wc_demo
[...lots of status information from Spark...]
[klevin@flux-hadoop-login2 pyspark_demo]$ hdfs dfs -ls wc_demo/
Found 3 items
-rw-r--r--   3 klevin hdfs        0 2019-03-12 11:58 wc_demo/_SUCCESS
-rw-r--r--   3 klevin hdfs        94 2019-03-12 11:58 wc_demo/part-00000
-rw-r--r--   3 klevin hdfs        108 2019-03-12 11:58 wc_demo/part-00001
[klevin@flux-hadoop-login2 pyspark_demo]$ hdfs dfs -cat wc_demo/*
('this', 2)
('is', 1)
('just', 1)
[...]
('hdfs.', 1)
[klevin@flux-hadoop-login2 pyspark_demo]$
```

Submitting to the queue: `spark-submit`

```
[klevin@flux-hadoop-login2 pyspark_demo]$ spark-submit --master yarn
--queue stats507w19 ps_wordcount.py /var/stats507w19/demo_file.txt wc_demo
[...lots of status information from Spark...]
[klevin@flux-hadoop-login2 pyspark_demo]$ hdfs dfs -ls wc_demo/
Found 3 items
-rw-r--r--   3 klevin hdfs          0 2019-03-12 11:58 wc_demo/_SUCCESS
-rw-r--r--   3 klevin hdfs          94 2019-03-12 11:58 wc_demo/part-00000
-rw-r--r--   3 klevin hdfs          108 2019-03-12 11:58 wc_demo/part-00001
[klevin@flux-hadoop-login2 pyspark_demo]$ hdfs dfs -cat wc_demo/*
('this', 2)
('is', 1)
('just', 1)
[...]
('hdfs.', 1)
[klevin@flux-hadoop-login2
```

Note: this status information may include some errors caused by the fact that some of the Python scripts that pyspark calls are still using the Python2 print statements. This causes Python3 to throw an error, which you can safely ignore. I have alerted IT to the issue.

Submitting to the queue: `spark-submit`



```
[klevin@flux-hadoop-login2 pyspark_demo]$ spark-submit --master yarn
--queue stats507w19 ps_wordcount.py /var/stats507w19/demo_file.txt wc_demo
[...lots of status information from Spark...]
[klevin@flux-hadoop-login2 pyspark_demo]$ hdfs dfs -ls wc_demo/
Found 3 items
-rw-r--r--   3 klevin hdfs          0 2019-03-12 11:58 wc_demo/_SUCCESS
-rw-r--r--   3 kle                                                 00000
-rw-r--r--   3 kle                                                -00001
[klevin@flux-hadoo
('this', 2)
('is', 1)
('just', 1)
[...]
('hdfs.', 1)
[klevin@flux-hadoop-login2 pyspark_demo]$
```

Specifying the master and queue are mandatory, but there are other additional options we could supply. Most importantly:

```
--num-executors 35
--executor-memory 5g
--executor-cores 4
```

More: https://spark.apache.org/docs/latest/submitting-applications.html

Submitting to the queue: `spark-submit`

Larger-scale example (runs on all of Google ngrams):

    http://arc-ts.umich.edu/hadoop-user-guide/#document-5

Warning: make sure you provide enough executors or this will take a long time!

Shared Variables

You won't need these in your homework, but they're extremely useful for more complicated jobs, especially ones that are not embarrassingly parallel.

Spark supports shared variables!

Allows for (limited) communication between parallel jobs

Two types:

- **Broadcast variables:** used to communicate value to all nodes

- **Accumulators:** nodes can only "add"
  - (or multiply, or… any operation on a **monoid**)
    - https://en.wikipedia.org/wiki/Monoid
  - https://spark.apache.org/docs/latest/rdd-programming-guide.html#accumulators