STATS 701
Data Analysis using Python

Lecture 20: Hadoop and the `mrjob` package

Some slides adapted from C. Budak

Recap

Previous lecture: Hadoop/MapReduce framework in general

This lecture: actually doing things

In particular: `mrjob` Python package

- https://pythonhosted.org/mrjob/
- **Installation:** `pip install mrjob` (or conda, or install from source...)

Recap: Basic concepts

Mapper: takes a (key,value) pair as input

Outputs zero or more (key,value) pairs

Outputs grouped by key

Combiner: takes a key and a subset of values for that key as input

Outputs zero or more (key,value) pairs

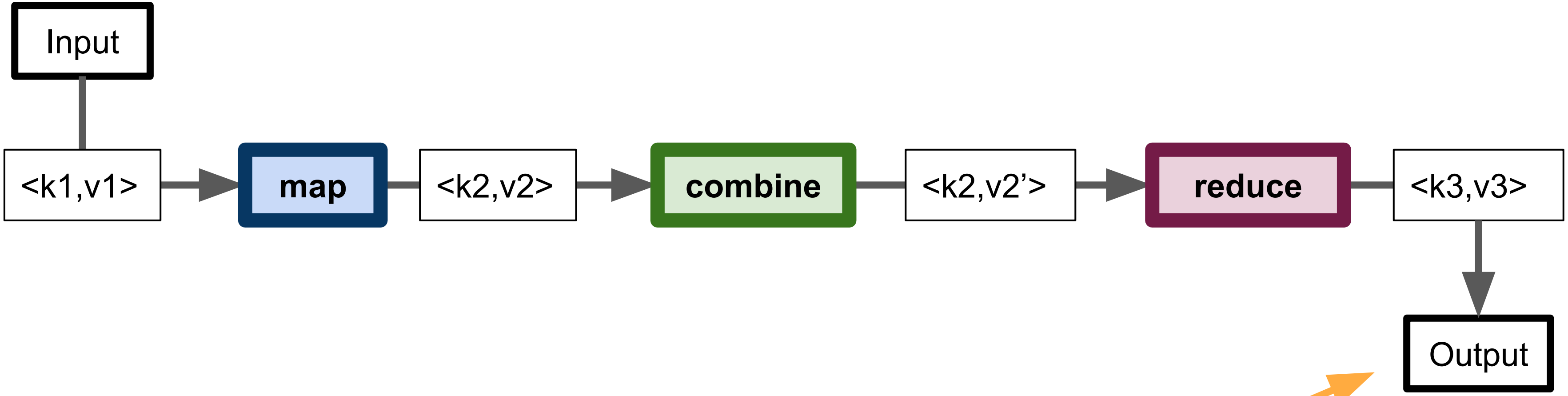
Runs after the mapper, only on a slice of the data

Must be **idempotent**

Reducer: takes a key and **all** values for that key as input

Outputs zero or more (key,value) pairs

Recap: a prototypical MapReduce program



Note: this output could be made the input to another MR program.

Recap: Basic concepts

Step: One sequence of map, combine, reduce

All three are optional, but must have at least one!

Node: a computing unit (e.g., a server in a rack)

Job tracker: a single node in charge of coordinating a Hadoop job

Assigns tasks to worker nodes

Worker node: a node that performs actual computations in Hadoop

e.g., computes the Map and Reduce functions

Python `mrjob` package

Developed at Yelp for simplifying/prototyping MapReduce jobs
https://engineeringblog.yelp.com/2010/10/mrjob-distributed-computing-for-everybody.html

`mrjob` acts like a wrapper around **Hadoop Streaming**

Hadoop Streaming makes Hadoop computing model available to languages other than Java

But `mrjob` can also be run without a Hadoop instance at all!

e.g., locally on your machine

Why use `mrjob`?

Fast prototyping

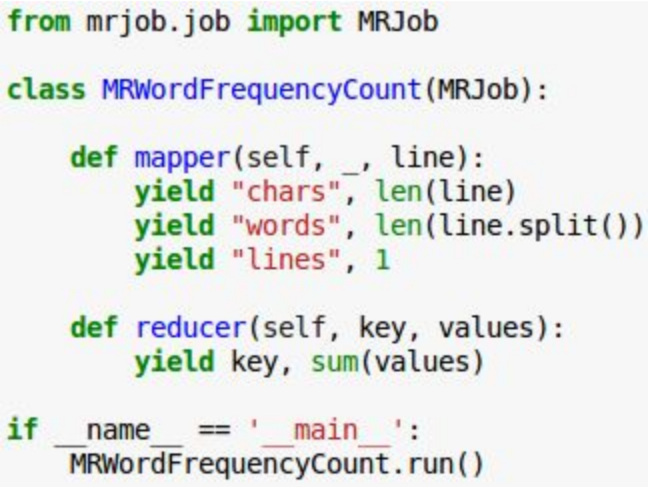
- Can run locally without a Hadoop instance...
- ...but can also run atop Hadoop or Spark

Much simpler interface than Java Hadoop

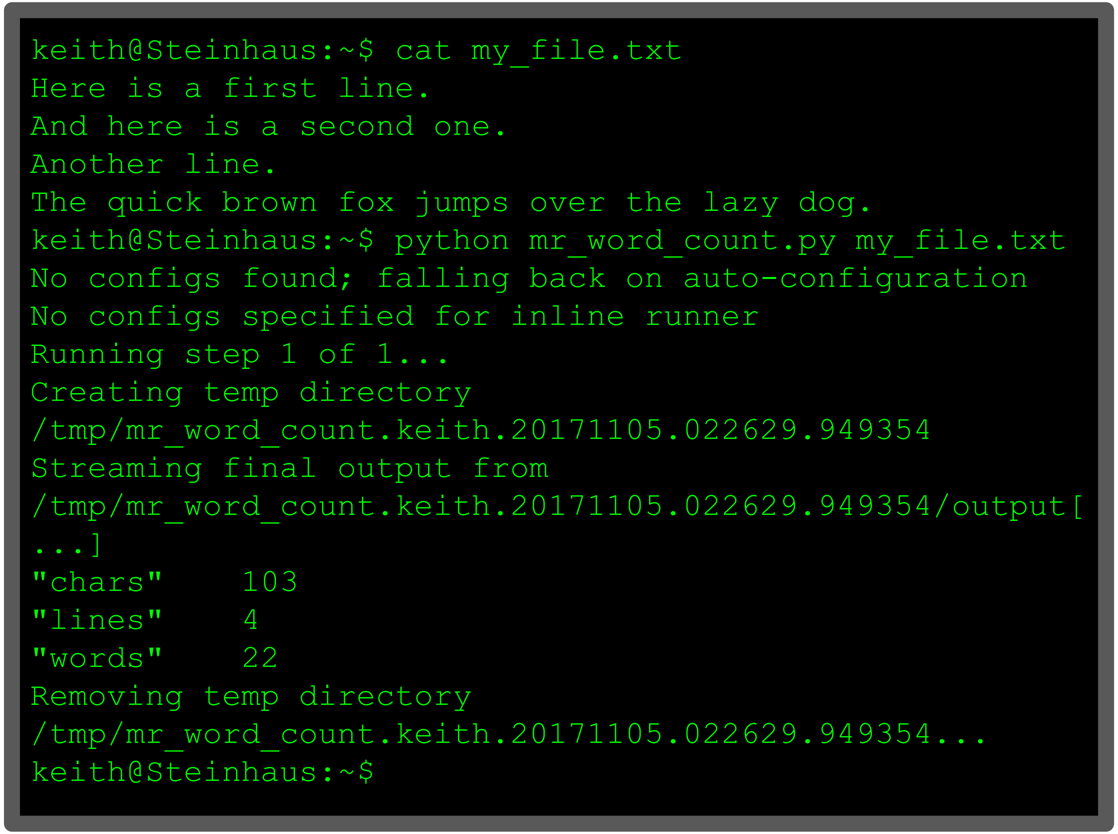
Sensible error messages

- i.e., usually there’s a Python traceback error if something goes wrong
- Because everything runs “in Python”

Basic `mrjob` script



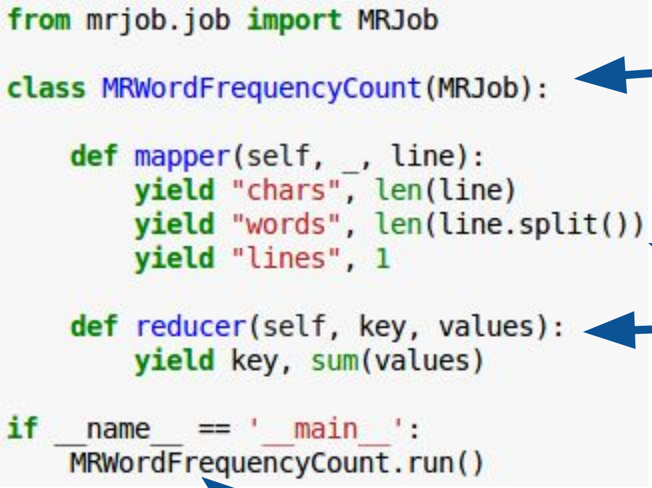
```python
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    MRWordFrequencyCount.run()
```



```
keith@Steinhaus:~$ cat my_file.txt
Here is a first line.
And here is a second one.
Another line.
The quick brown fox jumps over the lazy dog.
keith@Steinhaus:~$ python mr_word_count.py my_file.txt
No configs found; falling back on auto-configuration
No configs specified for inline runner
Running step 1 of 1...
Creating temp directory
/tmp/mr_word_count.keith.20171105.022629.949354
Streaming final output from
/tmp/mr_word_count.keith.20171105.022629.949354/output[
...]
"chars"    103
"lines"    4
"words"    22
Removing temp directory
/tmp/mr_word_count.keith.20171105.022629.949354...
keith@Steinhaus:~$
```

Basic `mrjob` script

This is a MapReduce job that counts the number of characters, words, and lines in a file.



```python
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

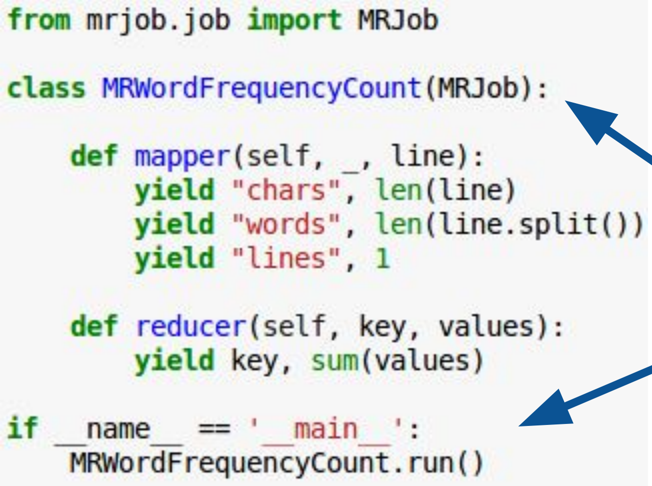
Each mrjob program you write requires defining a class, which extends the MRJob class.

These mapper and reducer methods are precisely the Map and Reduce operations in our job. Recall the difference between the **yield** keyword and the **return** keyword.

This if-statement will run precisely when we call this script from the command line.

Basic `mrjob` script

This is a MapReduce job that counts the number of characters, words, and lines in a file.

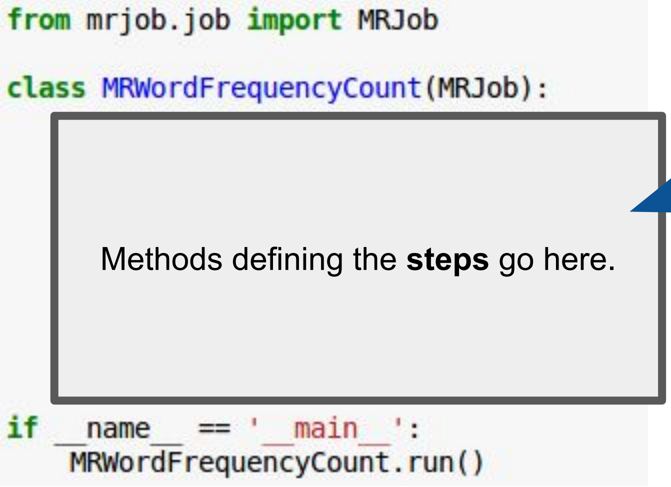


```
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

MRJob class already provides a method `run()`, which MRWordFrequencyCount inherits, but we need to define at least one of `mapper`, `reducer` or `combiner`.

This if-statement will run precisely when we call this script from the command line.

Basic `mrjob` script



```
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):
```

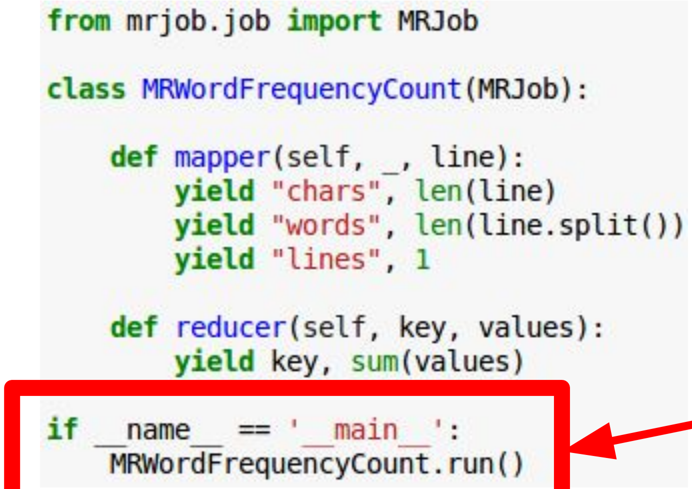
Methods defining the **steps** go here.

```
if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

In `mrjob`, an MRJob object implements one or more steps of a MapReduce program. Recall that a step is a single Map->Reduce->Combine chain. All three are optional, but must have at least one in each step.

If we have more than one step, then we have to do a bit more work… (we’ll come back to this)

Basic `mrjob` script

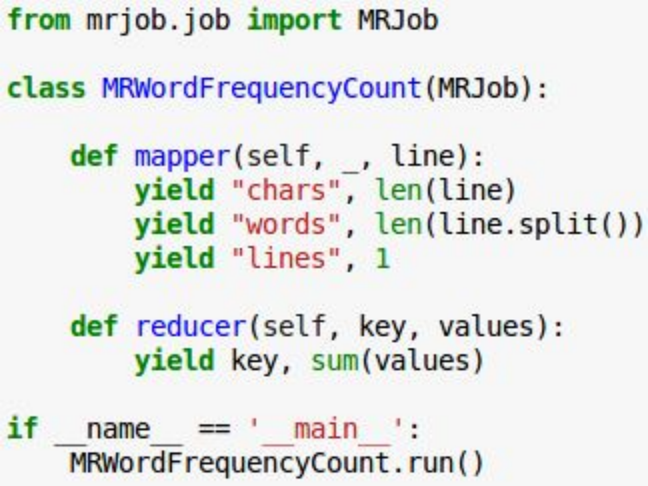


```python
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    MRWordFrequencyCount.run()
```

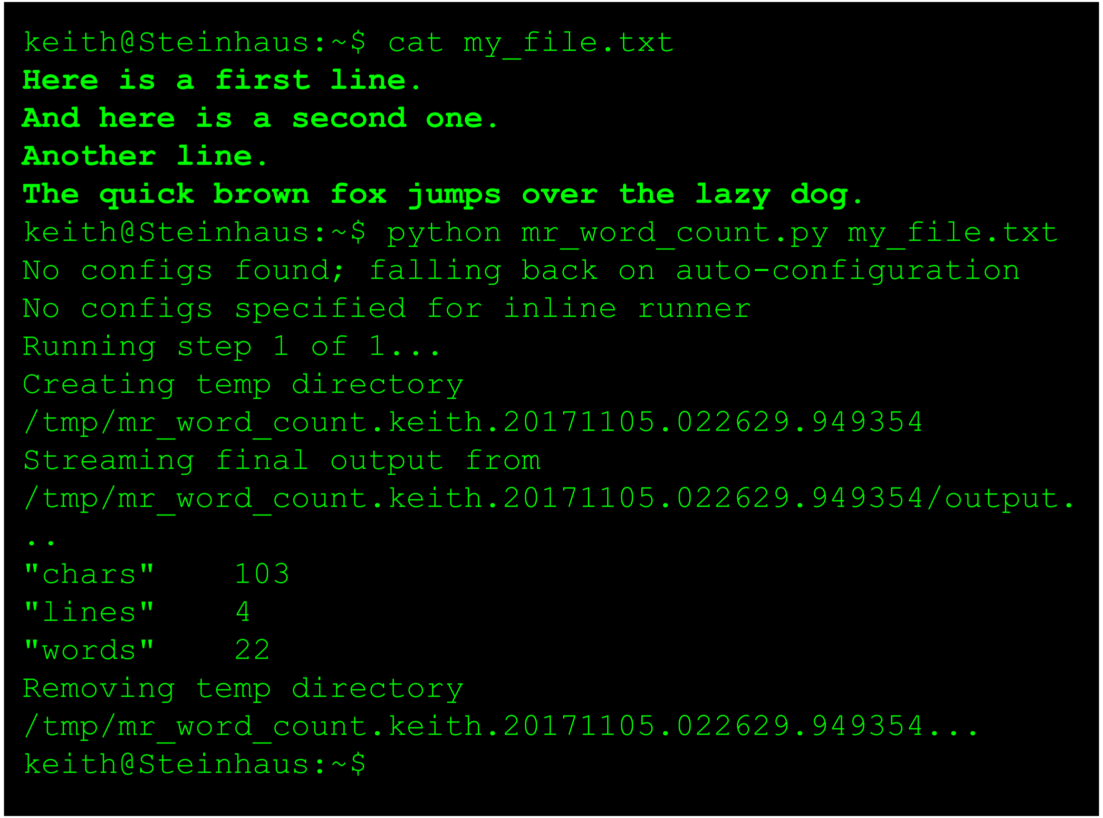
This is a MapReduce job that counts the number of characters, words, and lines in a file.

Warning: do not forget these two lines, or else your script will not run!

Basic `mrjob` script: recap

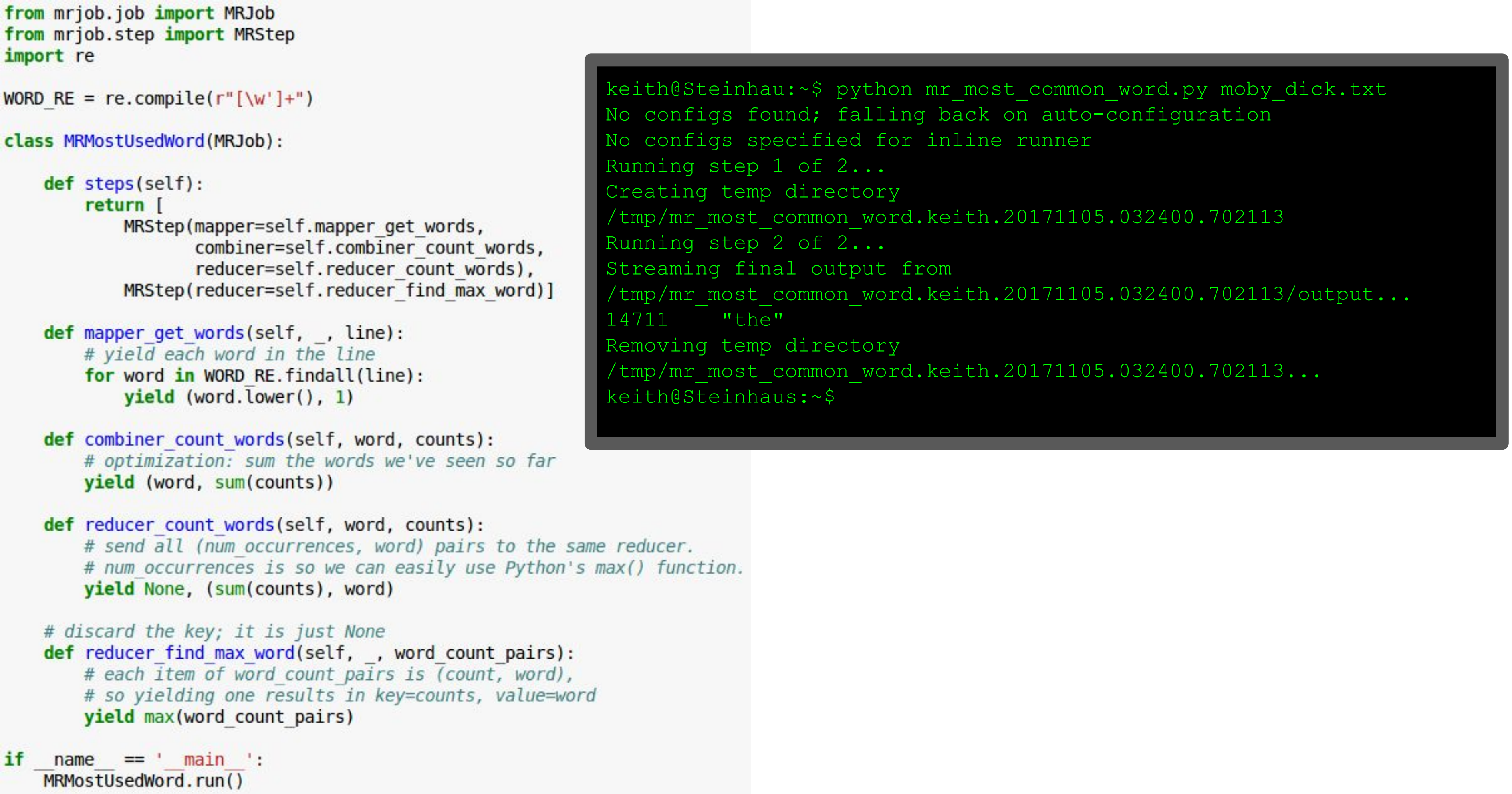


```python
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    MRWordFrequencyCount.run()
```



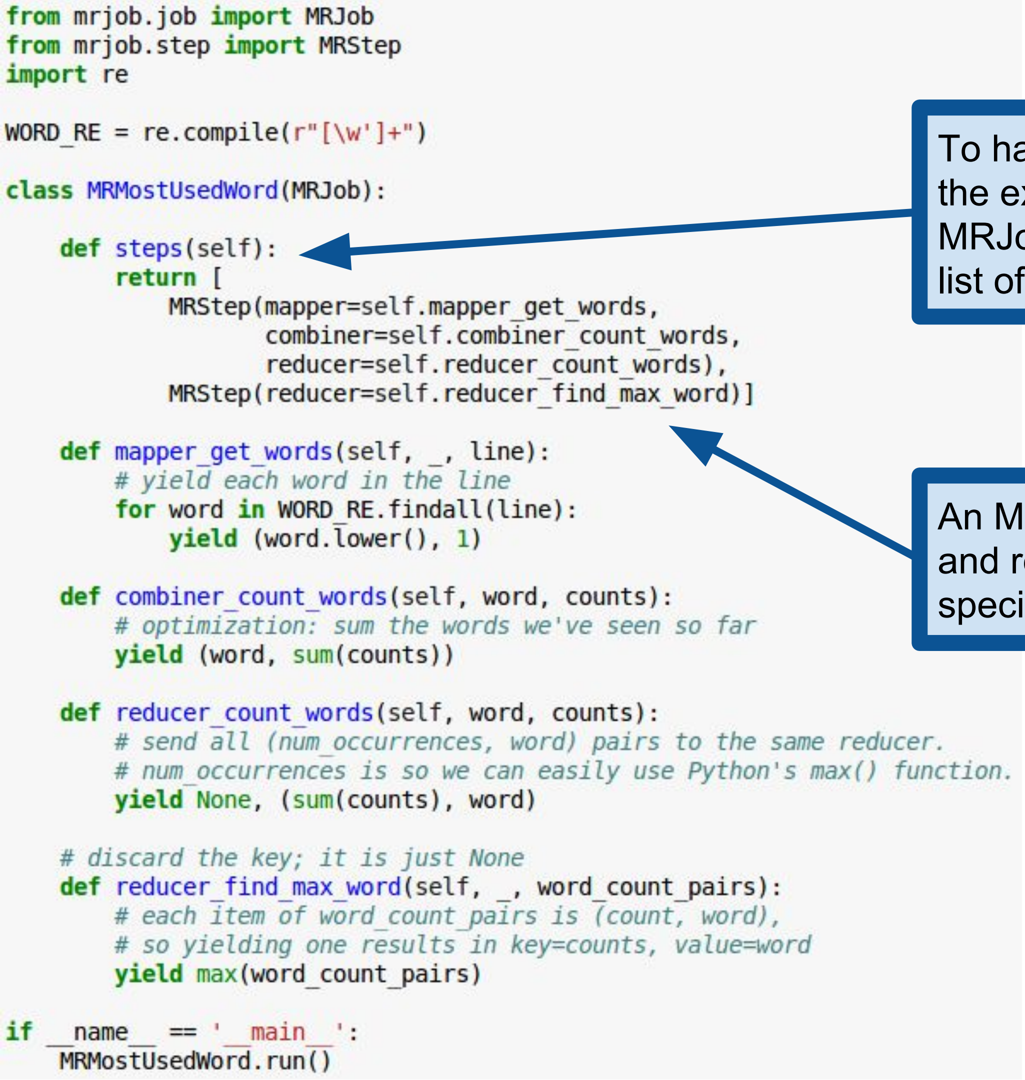
```
keith@Steinhaus:~$ cat my_file.txt
Here is a first line.
And here is a second one.
Another line.
The quick brown fox jumps over the lazy dog.
keith@Steinhaus:~$ python mr_word_count.py my_file.txt
No configs found; falling back on auto-configuration
No configs specified for inline runner
Running step 1 of 1...
Creating temp directory
/tmp/mr_word_count.keith.20171105.022629.949354
Streaming final output from
/tmp/mr_word_count.keith.20171105.022629.949354/output.
..
"chars"    103
"lines"    4
"words"    22
Removing temp directory
/tmp/mr_word_count.keith.20171105.022629.949354...
keith@Steinhaus:~$
```

More complicated jobs: multiple steps



```python
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w']+")

class MRMostUsedWord(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)]

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        # optimization: sum the words we've seen so far
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
        yield None, (sum(counts), word)

    # discard the key; it is just None
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
        yield max(word_count_pairs)

if __name__ == '__main__':
    MRMostUsedWord.run()
```

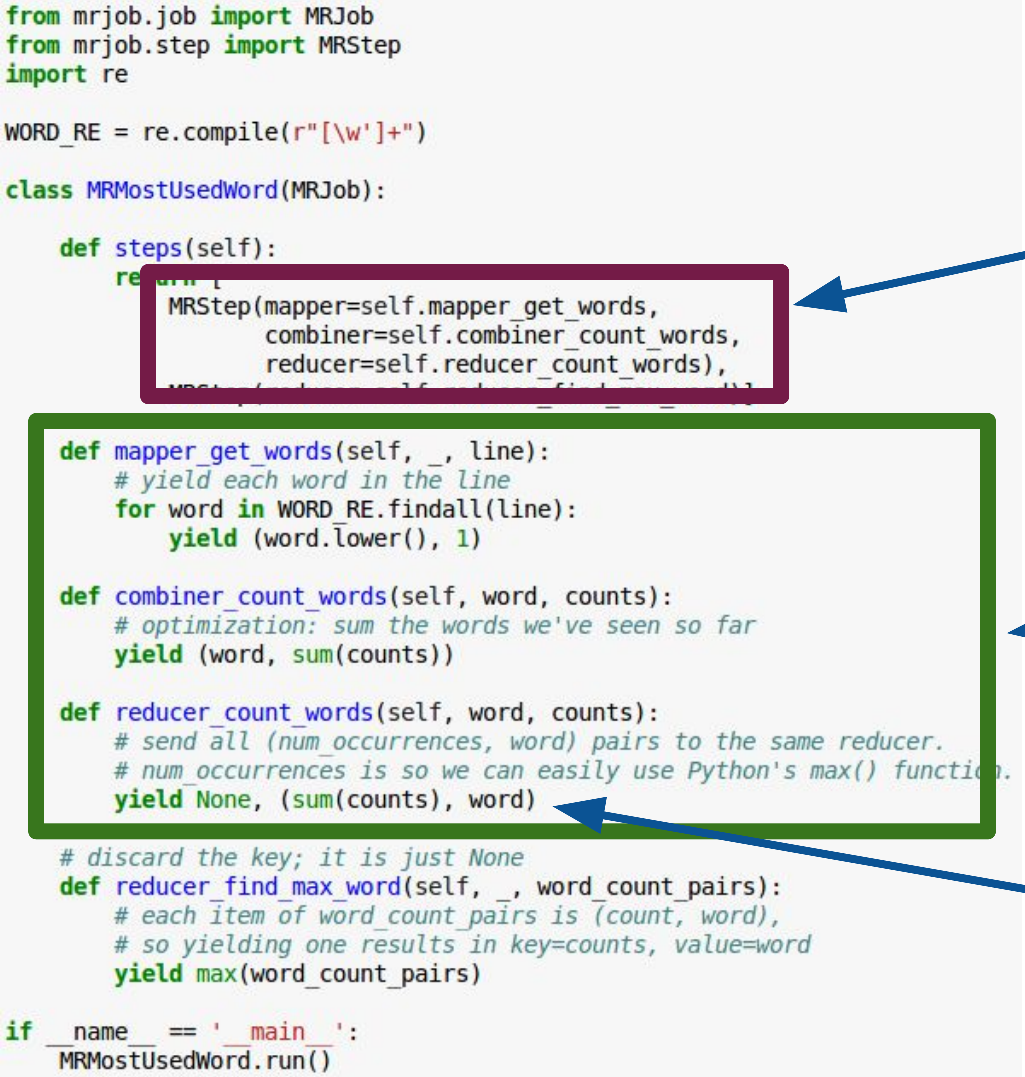
```
keith@Steinhau:~$ python mr_most_common_word.py moby_dick.txt
No configs found; falling back on auto-configuration
No configs specified for inline runner
Running step 1 of 2...
Creating temp directory
/tmp/mr_most_common_word.keith.20171105.032400.702113
Running step 2 of 2...
Streaming final output from
/tmp/mr_most_common_word.keith.20171105.032400.702113/output...
14711    "the"
Removing temp directory
/tmp/mr_most_common_word.keith.20171105.032400.702113...
keith@Steinhaus:~$
```



```python
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w']+")

class MRMostUsedWord(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)]

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        # optimization: sum the words we've seen so far
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
        yield None, (sum(counts), word)

    # discard the key; it is just None
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
        yield max(word_count_pairs)

if __name__ == '__main__':
    MRMostUsedWord.run()
```

To have more than one step, we need to override the existing definition of the method `steps()` in MRJob. The new `steps()` method must return a list of MRStep objects.

An MRStep object specifies a mapper, combiner and reducer. All three are optional, but must specify at least one.

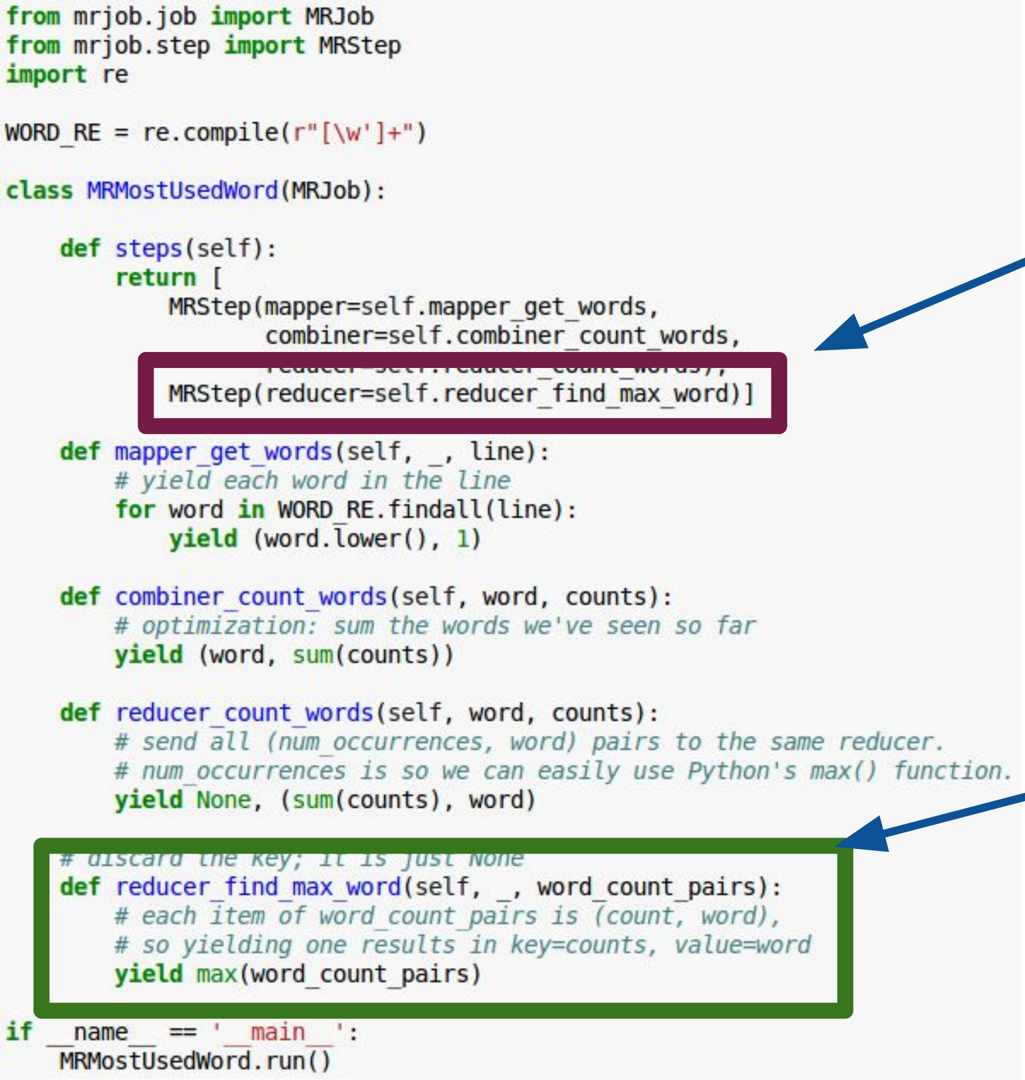


```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w']+")

class MRMostUsedWord(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)]

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        # optimization: sum the words we've seen so far
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
        yield None, (sum(counts), word)

    # discard the key; it is just None
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
        yield max(word_count_pairs)

if __name__ == '__main__':
    MRMostUsedWord.run()
```

First step: count words

This pattern should look familiar. It implements word counting.

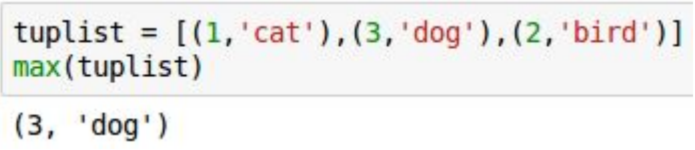
One key difference, because this reducer output is going to be the input to another step.



```python
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w']+")

class MRMostUsedWord(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)]

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        # optimization: sum the words we've seen so far
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
        yield None, (sum(counts), word)

    # discard the key; it is just None
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
        yield max(word_count_pairs)

if __name__ == '__main__':
    MRMostUsedWord.run()
```

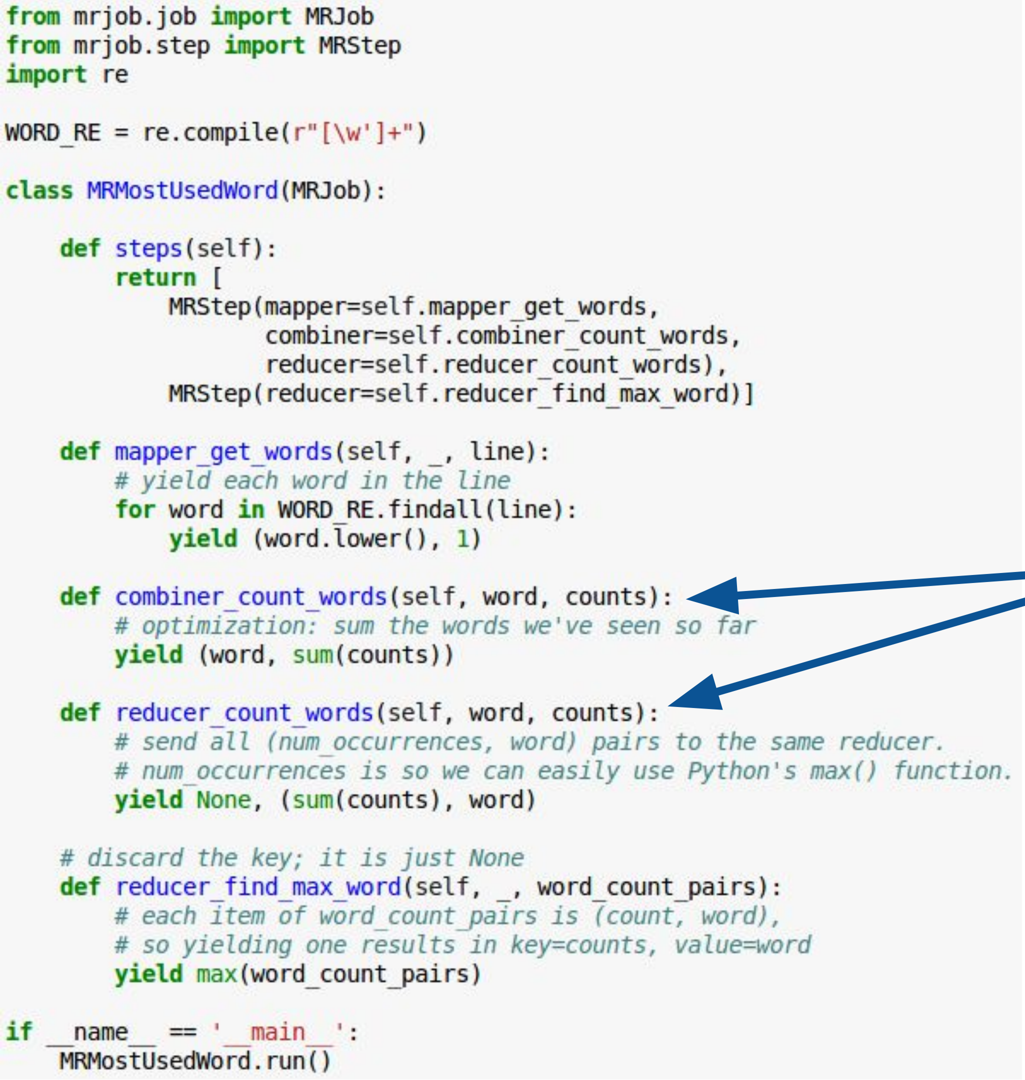
Second step: find the largest count.

Note: word_count_pairs is like a list of pairs. Refer to how Python max works on a list of tuples.



```python
tuplist = [(1,'cat'),(3,'dog'),(2,'bird')]
max(tuplist)
```

```
(3, 'dog')
```



```python
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w']+")

class MRMostUsedWord(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)]

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        # optimization: sum the words we've seen so far
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
        yield None, (sum(counts), word)

    # discard the key; it is just None
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
        yield max(word_count_pairs)

if __name__ == '__main__':
    MRMostUsedWord.run()
```

Note: combiner and reducer are the same operation in this example, provided we ignore the fact that reducer has a special output format

`MRJob.{mapper, combiner, reducer}`

MRJob.mapper(*key*, *value*)

key – parsed from input; **value** – parsed from input.

Yields zero or more tuples of (out_key, out_value).

MRJob.combiner(*key*, *values*)

key – yielded by mapper; **value** – generator yielding all values from node corresponding to key.

Yields one or more tuples of (out_key, out_value)

MRJob.reducer(*key*, *values*)

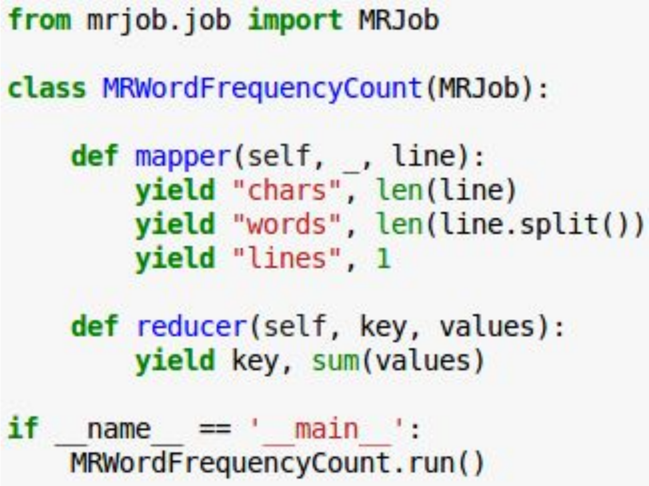
key – key yielded by mapper; **value** – generator yielding all values from corresponding to key.

Yields one or more tuples of (out_key, out_value)

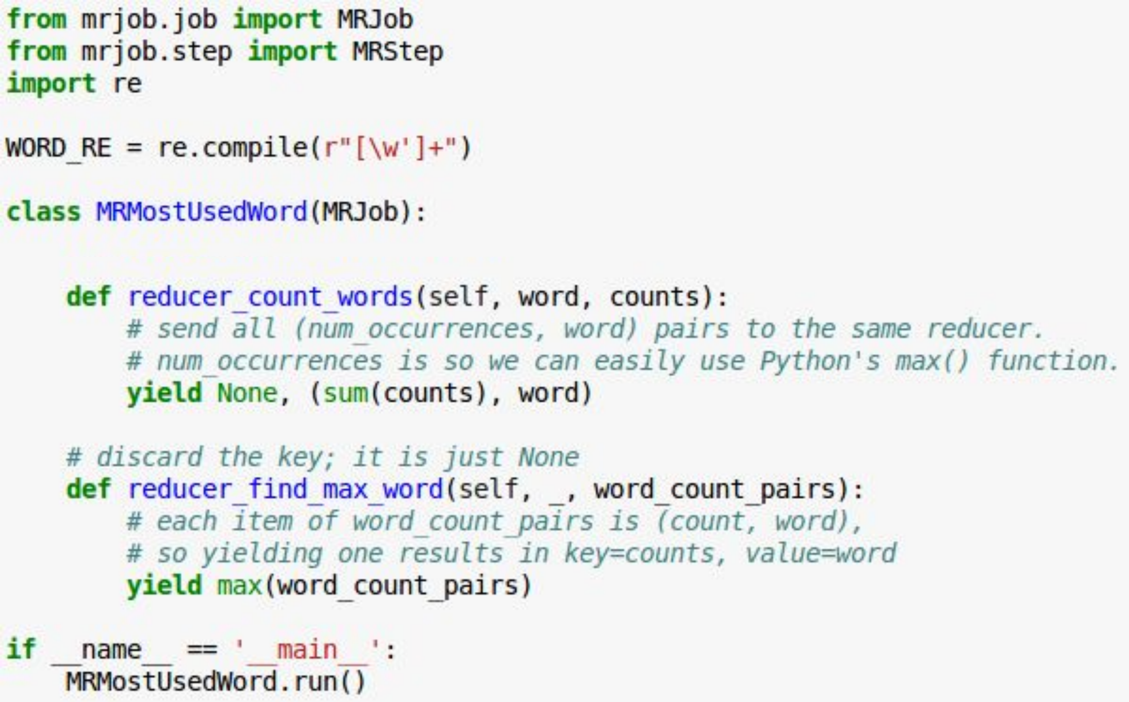
Details: https://pythonhosted.org/mrjob/job.html

More complicated reducers: Python’s `reduce`

So far our reducers have used Python built-in functions `sum` and `max`



```python
from mrjob.job import MRJob

class MRWordFrequencyCount(MRJob):

    def mapper(self, _, line):
        yield "chars", len(line)
        yield "words", len(line.split())
        yield "lines", 1

    def reducer(self, key, values):
        yield key, sum(values)

if __name__ == '__main__':
    MRWordFrequencyCount.run()
```



```python
from mrjob.job import MRJob
from mrjob.step import MRStep
import re

WORD_RE = re.compile(r"[\w']+")

class MRMostUsedWord(MRJob):


    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
        yield None, (sum(counts), word)

    # discard the key; it is just None
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
        yield max(word_count_pairs)

if __name__ == '__main__':
    MRMostUsedWord.run()
```

More complicated reducers: Python's `reduce`

So far our reducers have used Python built-in functions `sum` and `max`

What if I want to multiply the values instead of `sum`?

Python does not have `product()` function analogous to `sum()`...

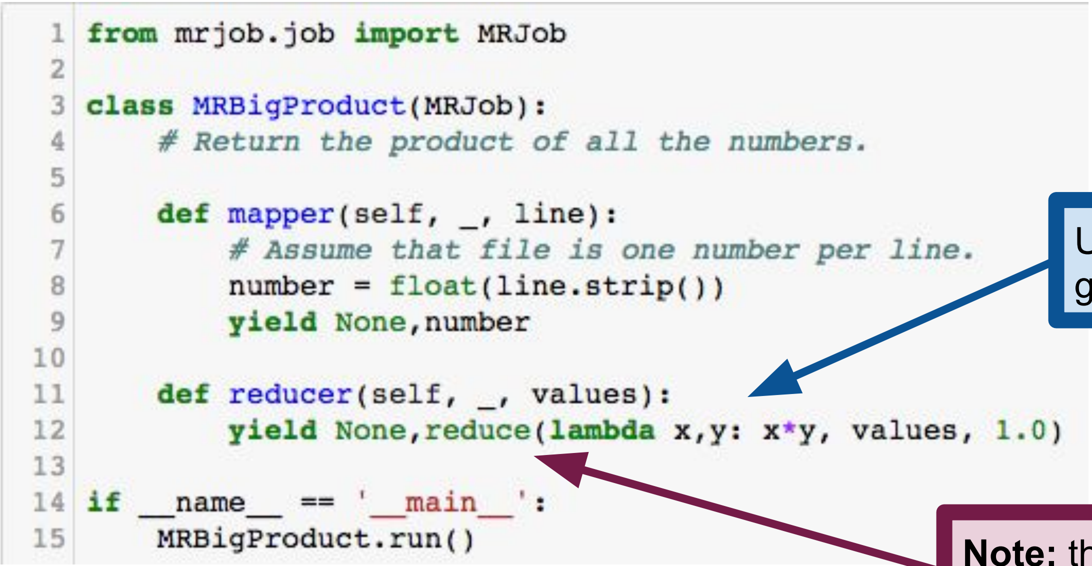
What if my values aren't numbers, but I have a sum defined on them?

e.g., tuples representing vectors

Want `(a,b)+(x,y)=(a+x,b+y)`, but tuples don't support addition

Solution: use `functools.reduce`

More complicated reducers: Python's `reduce`



```
from mrjob.job import MRJob

class MRBigProduct(MRJob):
    # Return the product of all the numbers.

    def mapper(self, _, line):
        # Assume that file is one number per line.
        number = float(line.strip())
        yield None,number

    def reducer(self, _, values):
        yield None,reduce(lambda x,y: x*y, values, 1.0)

if __name__ == '__main__':
    MRBigProduct.run()
```

Using `reduce` and `lambda`, we can get just about any reducer we want.

Note: this example was run in Python 2. You'll need to import `functools` to do this.

Running `mrjob` on a Hadoop cluster

We've already seen how to run mrjob from the command line.

- Previous examples emulated Hadoop
- But no actual Hadoop instance was running!

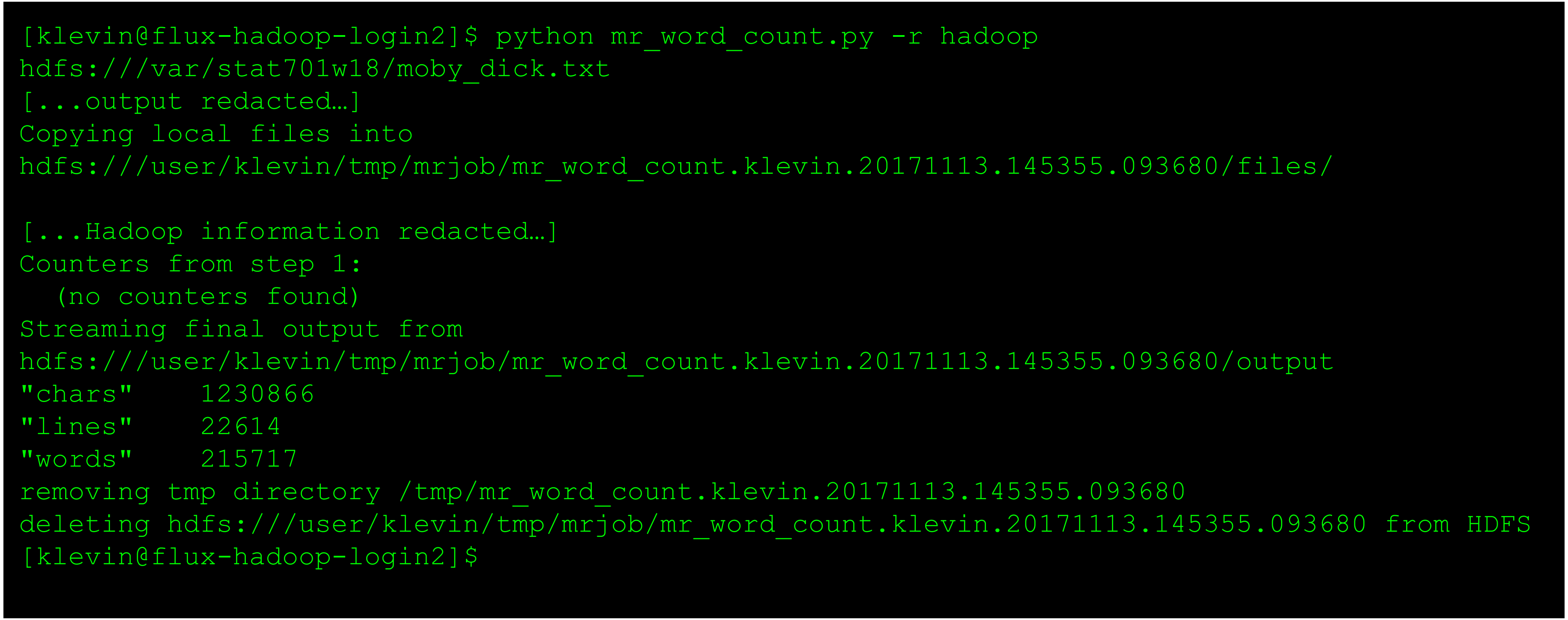
That's fine for prototyping and testing…

...but how do I actually run it on my Hadoop cluster?

- E.g., on Fladoop

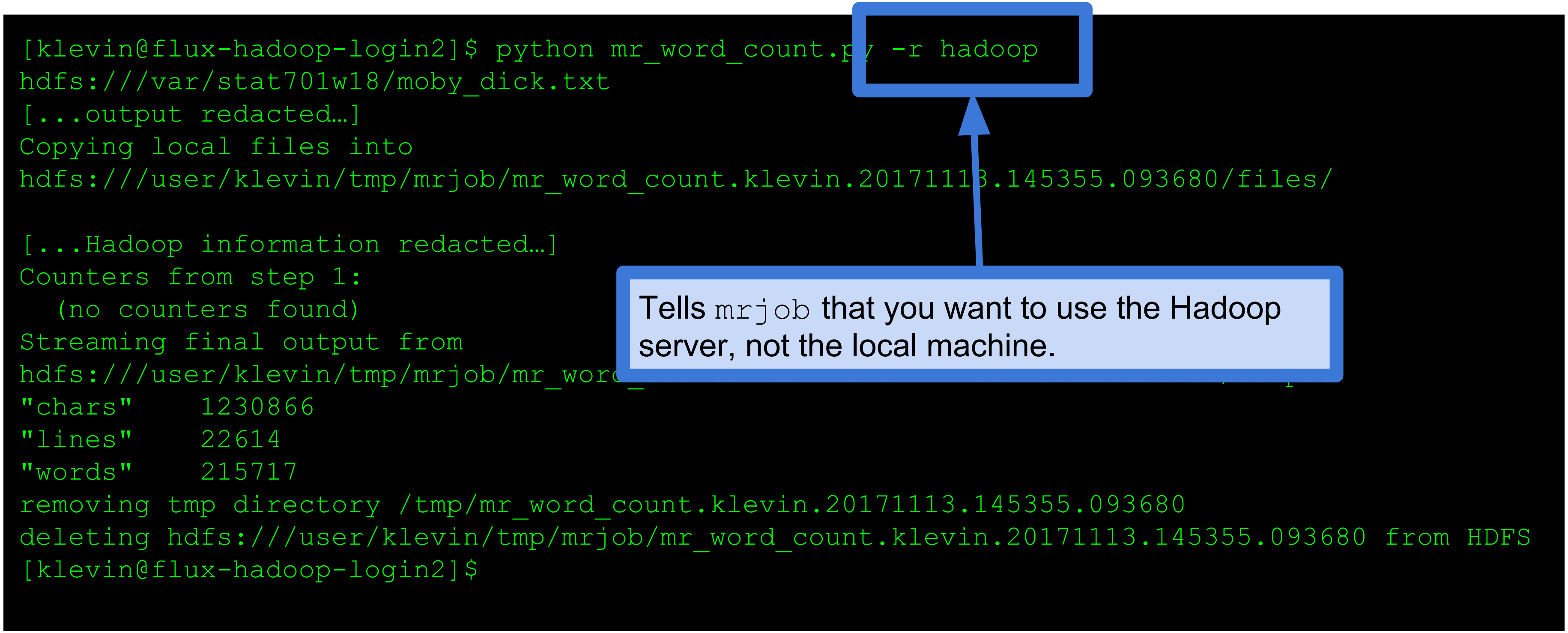
Open a terminal and sign on to Fladoop if you'd like to follow along.

Running `mrjob` on Fladoop



```
[klevin@flux-hadoop-login2]$ python mr_word_count.py -r hadoop
hdfs:///var/stat701w18/moby_dick.txt
[...output redacted…]
Copying local files into
hdfs:///user/klevin/tmp/mrjob/mr_word_count.klevin.20171113.145355.093680/files/

[...Hadoop information redacted…]
Counters from step 1:
  (no counters found)
Streaming final output from
hdfs:///user/klevin/tmp/mrjob/mr_word_count.klevin.20171113.145355.093680/output
"chars"    1230866
"lines"    22614
"words"    215717
removing tmp directory /tmp/mr_word_count.klevin.20171113.145355.093680
deleting hdfs:///user/klevin/tmp/mrjob/mr_word_count.klevin.20171113.145355.093680 from HDFS
[klevin@flux-hadoop-login2]$
```

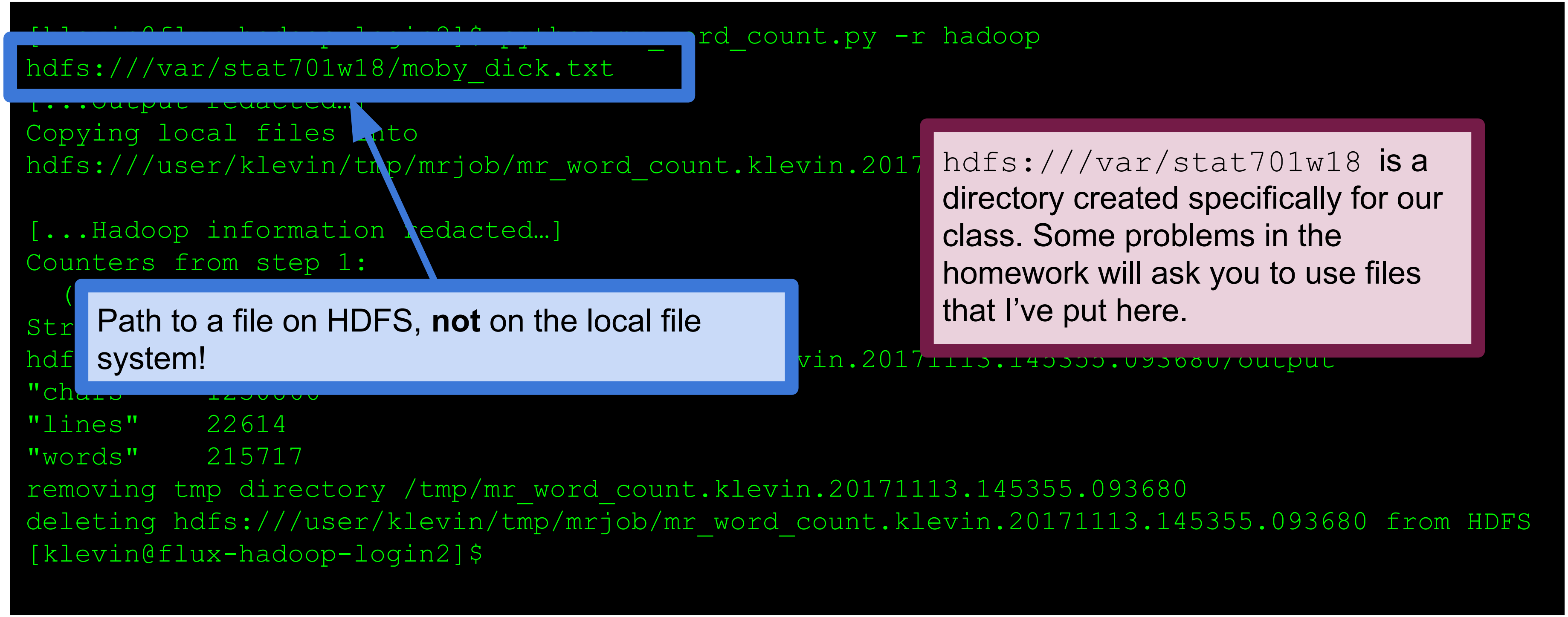
Running mrjob on Fladoop



```
[klevin@flux-hadoop-login2]$ python mr_word_count.py -r hadoop
hdfs:///var/stat701w18/moby_dick.txt
[...output redacted…]
Copying local files into
hdfs:///user/klevin/tmp/mrjob/mr_word_count.klevin.20171113.145355.093680/files/

[...Hadoop information redacted…]
Counters from step 1:
  (no counters found)
Streaming final output from
hdfs:///user/klevin/tmp/mrjob/mr_word_
"chars"    1230866
"lines"    22614
"words"    215717
removing tmp directory /tmp/mr_word_count.klevin.20171113.145355.093680
deleting hdfs:///user/klevin/tmp/mrjob/mr_word_count.klevin.20171113.145355.093680 from HDFS
[klevin@flux-hadoop-login2]$
```

Tells mrjob that you want to use the Hadoop server, not the local machine.

Running `mrjob` on Fladoop



```
[klevin@flux-hadoop-login2]$ python mr_word_count.py -r hadoop
hdfs:///var/stat701w18/moby_dick.txt
[...output redacted…]
Copying local files into
hdfs:///user/klevin/tmp/mrjob/mr_word_count.klevin.2017

[...Hadoop information redacted…]
Counters from step 1:
   (
Str
hdf                                         vin.20171113.145355.093680/output
"chars"     1230866
"lines"     22614
"words"     215717
removing tmp directory /tmp/mr_word_count.klevin.20171113.145355.093680
deleting hdfs:///user/klevin/tmp/mrjob/mr_word_count.klevin.20171113.145355.093680 from HDFS
[klevin@flux-hadoop-login2]$
```

Path to a file on HDFS, **not** on the local file system!

`hdfs:///var/stat701w18` is a directory created specifically for our class. Some problems in the homework will ask you to use files that I've put here.

HDFS is a separate file system

Local file system
Accessible via ls, mv, cp, cat...

/home/klevin

/home/klevin/stats701

/home/klevin/myfile.txt

(and lots of other files…)

Hadoop distributed file system
Accessible via hdfs...

/var/stat701w18

/var/stat701w18/fof

/var/stat701w18/populations_small.txt

(and lots of other files…)

Shell provides commands for moving files around, listing files, creating new files, etc. But if you try to use these commands to do things on HDFS... no dice!

Hadoop has a special command line tool for dealing with HDFS, called `hdfs`

Basics of `hdfs`

Usage: `hdfs dfs [options] COMMAND [arguments]`

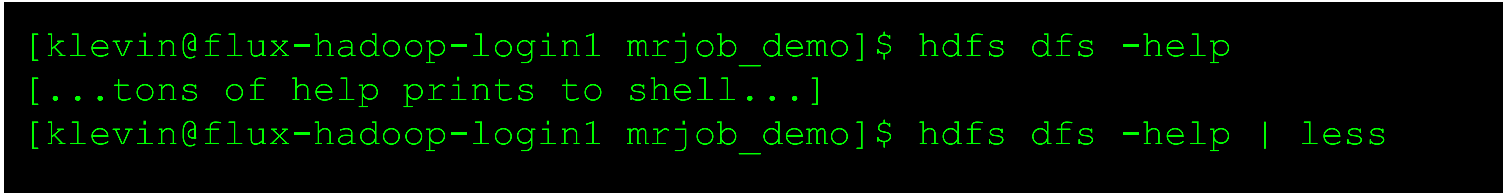
Where `COMMAND` is, for example:

`-ls`, `-mv`, `-cat`, `-cp`, `-put`, `-tail`

All of these should be pretty self-explanatory except `-put`

For your homework, you should only need `-cat` and perhaps `-cp`/`-put`

Getting help:



```
[klevin@flux-hadoop-login1 mrjob_demo]$ hdfs dfs -help
[...tons of help prints to shell...]
[klevin@flux-hadoop-login1 mrjob_demo]$ hdfs dfs -help | less
```

`hdfs` essentially replicates shell command line

```
[klevin@.]$ hdfs dfs -put demo_file.txt hdfs:///var/stat701w18/demo_file.txt
[klevin@.]$ hdfs dfs -cat hdfs:///var/stat701w18/demo_file.txt
This is just a demo file.
Normally, a file this small would have no reason to be on HDFS.
[klevin@.]$
```

Important points:

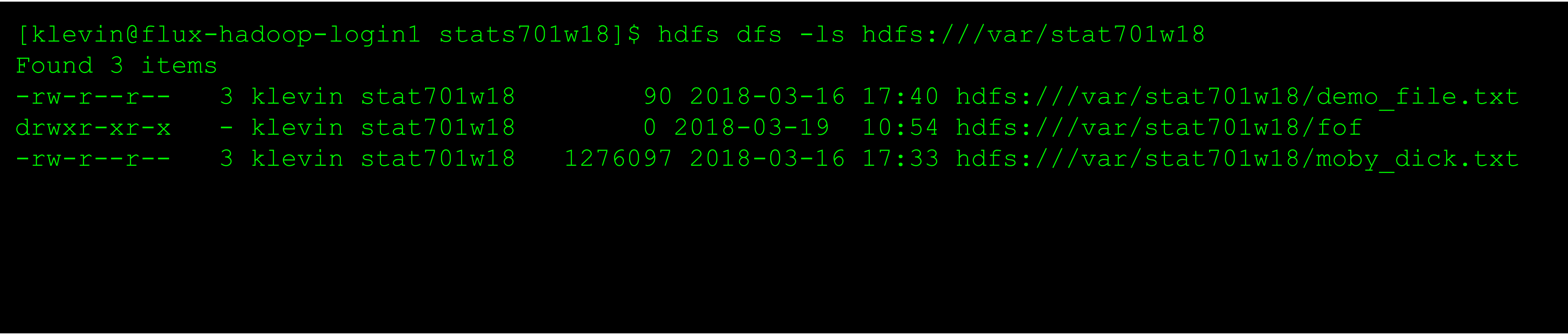
Note **three** slashes in `hdfs:///var/`…

`hdfs:///var` and `/var` are **different directories** on **different file systems**

`hdfs` **`dfs`** `-CMD` because hdfs supports lots of other stuff, too

Don’t forget a hyphen before your command! `-cat`, not `cat`

To see all our HDFS files



```
[klevin@flux-hadoop-login1 stats701w18]$ hdfs dfs -ls hdfs:///var/stat701w18
Found 3 items
-rw-r--r--   3 klevin stat701w18         90 2018-03-16 17:40 hdfs:///var/stat701w18/demo_file.txt
drwxr-xr-x   - klevin stat701w18        0 2018-03-19  10:54 hdfs:///var/stat701w18/fof
-rw-r--r--   3 klevin stat701w18    1276097 2018-03-16 17:33 hdfs:///var/stat701w18/moby_dick.txt
```

You'll use some of these files in your homework.

`mrjob` hides complexity of MapReduce

We need only define mapper, reducer, combiner

Package handles everything else

- Most importantly, interacting with Hadoop

But `mrjob` does provide powerful tools for specifying Hadoop configuration

- https://pythonhosted.org/mrjob/guides/configs-basics.html

> You don’t have to worry about any of this in this course, but you should be aware of it in case you need it in the future.

`mrjob`: protocols

`mrjob` assumes that all data is “newline-delimited bytes”

- That is, newlines separate lines of input
- Each line is a single unit to be processed in isolation
  - (e.g., a line of words to count, an entry in a database, etc)

`mrjob` handles inputs and outputs via **protocols**

- **Protocol** is an object that has `read()` and `write()` methods
- `read()`: convert bytes to (key,value) pairs
- `write()`: convert (key,value) pairs to bytes

`mrjob`: protocols

Controlled by setting three variables in config file `mrjob.conf`:

INPUT_PROTOCOL, INTERNAL_PROTOCOL, OUTPUT_PROTOCOL

Defaults:

```
INPUT_PROTOCOL = mrjob.protocol.RawValueProtocol
INTERNAL_PROTOCOL = mrjob.protocol.JSONProtocol
OUTPUT_PROTOCOL = mrjob.protocol.JSONProtocol
```

Again, you don’t have to worry about this in this course, but you should be aware of it.

Data passed around internally via JSON. This is precisely the kind of thing that JSON is good for.

Readings

Required:

`mrjob` Fundamentals and Concepts

https://pythonhosted.org/mrjob/guides/quickstart.html

https://pythonhosted.org/mrjob/guides/concepts.html

Hadoop wiki: How MapReduce operations are actually carried out

https://wiki.apache.org/hadoop/HadoopMapReduce