MRJob

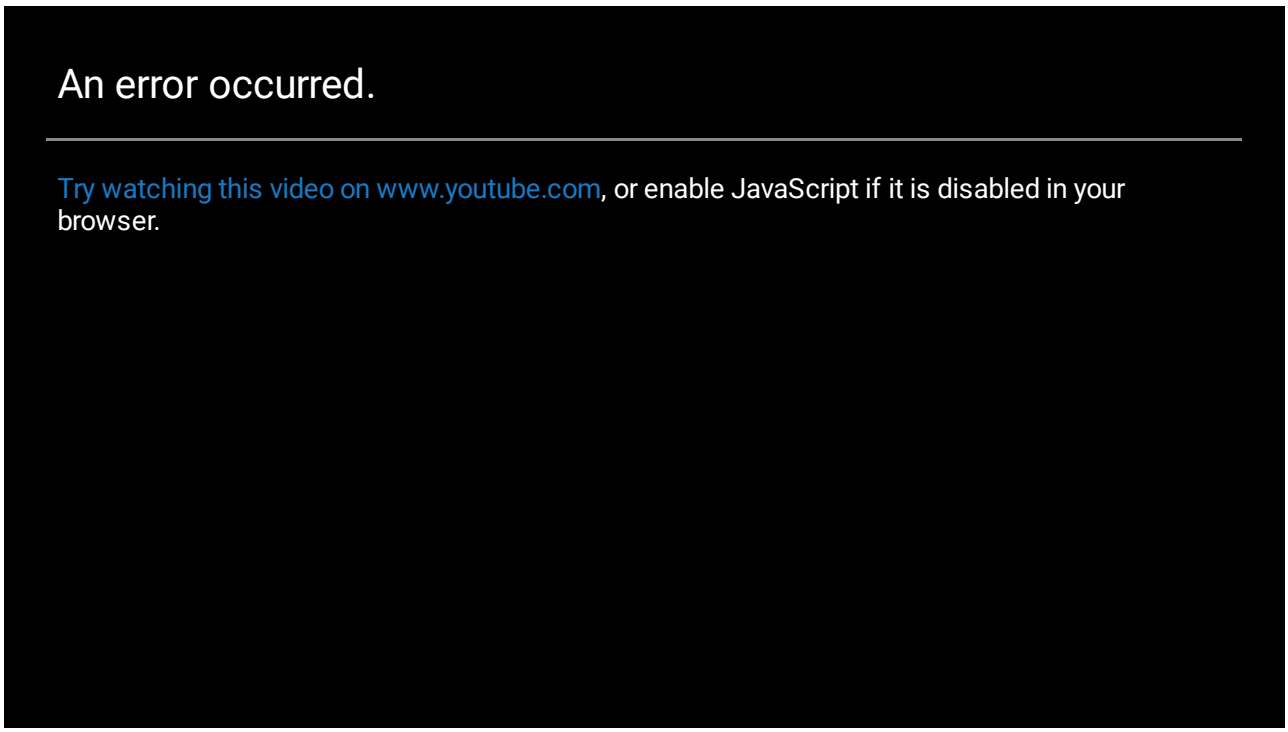
MRJob

You've now had some experience writing mappers and reducers to extract data from files stored in HDFS. As you will have noticed, it can be a bit laborious.

The company Yelp developed a Python library called "mrjob" to help with this. It contains a class called "MRJob" which simplifies the process of writing and running mapping and reducing functions in Python. All of your Python code can go into just a single file. And you can run your code locally without installing Hadoop, which is great for testing, or you can run it using Hadoop, either in standalone mode or on a cluster.

The best way to see how MRJob works is to work through some examples, and this lesson contains a number of such examples. Before we look at one, there is a video in the next slide which provides a helpful introduction to MRJob.

Introduction to MRJob (6:31)



Word frequency (Local)

Now let's see an example in action.

Consider again the task of **counting the frequency of words** in a file.

When you click the panel on the right you'll get a connection to a server that has a Python program called "job.py" in your home directory. This program uses MRJob to find the frequency of words in a file.

Here is a walk-through of the program:

First, we import the MRJob class from the mrjob.job module.

Next, we define a class that inherits from the MRJob class. Here we have called it 'Job', but you can call it whatever you like.

We define this class by defining a mapper method and a reducer method.

The `mapper` method takes a key and a value as arguments, and yields as many key-value pairs as we like. In this case, we ignore the key; the value is a single line of text, and for each word in the line we yield the pair `word, 1`.

The `reducer` method takes a key and a collection of values, and also yields as many key-value pairs as we like. In this case, the key is a word and the values are the frequencies of that word returned by the mappers; we yield the word and the sum of those frequencies, which is the total frequency of the word.

Finally, we add the last two lines. These lines pass control over to the command line arguments and execution to mrjob. You must include them every time - without them your job will not work.

Running the job

On the server, there is also the file "text.txt", which contains some text (you can open the file to look at it). To count the frequency of words in this file, enter the following command into the terminal:

```
$ python job.py text.txt
```

You should see the results appear in the terminal.

You can pass multiple input files to the job by adding them to the enter of this command. Try passing text.txt twice, by entering the following command. You should get word frequencies that are twice the size (because the file is being counted twice).

```
$ python job.py text.txt text.txt
```

By default your job is running locally, not using Hadoop. This is good for writing and testing your code, because it is fast. Ultimately you want our job to run on a file stored in your Hadoop cluster. We'll see how to do that in the next slide.

Word frequency (Hadoop)

When you click the panel on the right you will get a terminal connection to a server which has Hadoop and YARN installed and running. It also has the text.txt file and the job.py MRJob program from the previous slide.

You can choose whether to run job.py locally on the server or instead of the server's Hadoop cluster.

Run locally

You've already seen how to run it locally:

```
$ python job.py < text.txt
```

Run on Hadoop

To run it on the Hadoop cluster use the following command instead:

```
$ python job.py -r hadoop < text.txt
```

Here you are adding the `-r` option to your command ("r" is for "runner"). You can use `-r inline` (the default), `-r local` (which simulates some features of Hadoop), or `-r hadoop` (which runs your job on Hadoop).

Note that you can observe that the results are different when running on Hadoop. The reason is that, Hadoop will perform shuffling and sorting for the mapper output. The mapper output is received by the reducers with orders, and thus the final results are sorted by keys. However, if you run MRJob codes locally, Hadoop is not involved in this procedure, and the mapper output is thus not sorted. Therefore, if your reducer procedure depends on the order of the received mapper output, you must run and test your codes on Hadoop to check the correctness.

Note that it is much slower on Hadoop. In the rest of these slides we'll just run jobs locally, which is good for learning and testing, since these jobs do not rely on the mapper output's order.

Run on Hadoop with HDFS file

In the last example we ran job.py on the Hadoop cluster but we passed it a local file (that is, a file stored on the server's local file directory). You call also pass it a file stored in HDFS. Let's see how to do that.

First, create your default directory in HDFS and copy the file to it:

```
$ hdfs dfs -mkdir -p /user/user
```

```
$ hdfs dfs -put text.txt
```

Now pass this file to job.py by using the following command:

```
$ python job.py -r hadoop hdfs:///user/user/text.txt
```

You can also store the output in HDFS by using the "-o" option in your command:

```
$ python job.py -r hadoop hdfs:///user/user/text.txt -o hdfs:///user/user/output
```

After your job completes, you can check the results in HDFS by:

```
$ hdfs dfs -cat output/p*
```

Run locally with HDFS file?

You can't do that. Try it:

```
$ python job.py hdfs:///user/user/text.txt
```

Testing your mapper and reducer

With MRJob you can test your `mapper()` and `reducer()` methods individually to check that each is working properly.

To test your mapper, add the --mapper option to your run command:

```
$ python job.py --mapper text.txt
```

Similarly, to test your reducer add the --reducer option. In this case, you won't be able to use text.txt as the input to your job, because your reducer method is expecting a different kind of file - it's expecting the results of your mapper.

You have a couple of options.

First, you can send the results of your mapper to an output file and then pass that output file to your job. You can call the output file whatever you want - let's call it "output.txt". Here are the commands you should use:

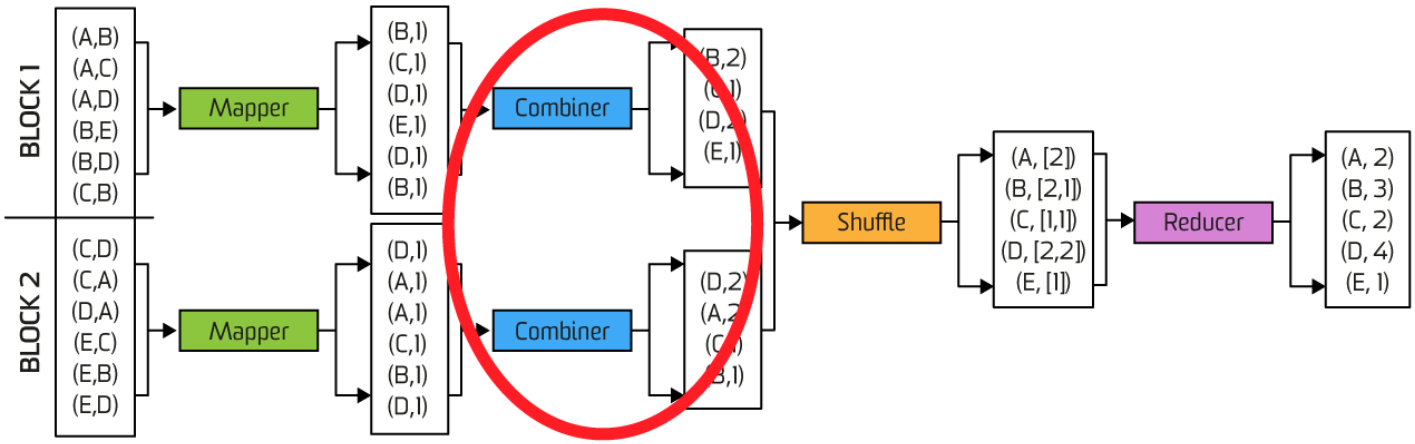
```
$ python job.py --mapper text.txt > output.txt
$ cat output.txt | sort -k1,1 | python job.py --reducer
```

Second, you can run your mapper and **pipe** the results to your reducer, with a single command:

```
$ python job.py --mapper text.txt | sort -k1,1 | python job.py --reducer
```

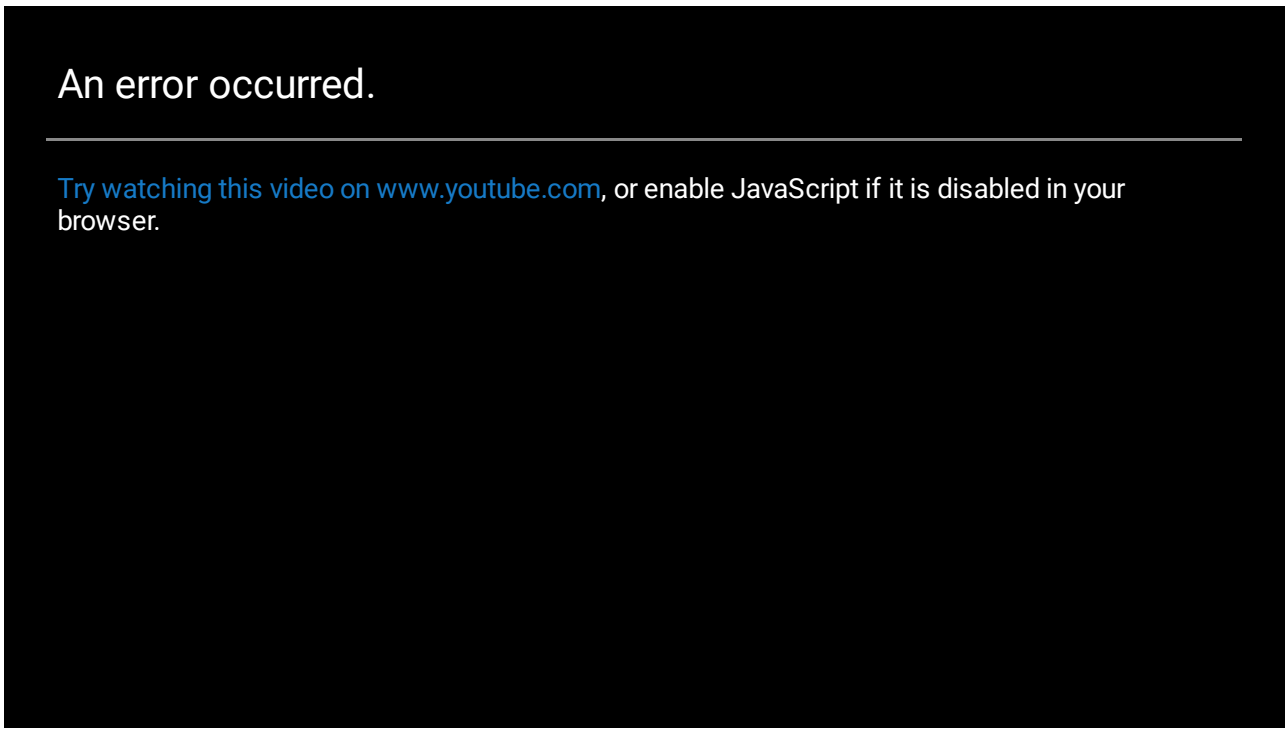
Using a combiner

Often a mapper function will produce many key-value pairs which have the same key (e.g. in the word frequency example : `"dog", 1`, `"dog", 1`, `"dog", 1`). These will ultimately get combined by the reducer function, but we can reduce the amount of intermediate data, thus saving network time, by combining the values of all keys on the same node before sending them to the reducer. We do this by defining a **combiner** function. Combiners can be thought of as mini-reducers.



We will see how to add a combiner to a MRJob program in the next coding slide.

The benefit of combiners (4:08)



Adding a combiner

Here is the word frequency job again, but with a combiner added.

Try running the job:

```
$ python job.py text.txt
```

Note that in this simple example, the combiner does the same thing as does the reducer. However, in most applications, you need to write a different combiner to locally aggregate the mapper output.

In this week's assessment, you will need to do a task about average value computation, and you are required to use a combiner for that task. The combiner and the reducer will be different in that task.

Example: file statistics

It will be helpful to go through some more examples.

When you click the panel on the right you'll get a connection to a server that has the file "text.txt" in your home directory.

There is also a MRJob program that calculates some basic statistics about a file: **the number of lines**, **the number of words**, and **the number of characters**. Read through the program and make sure you understand how it works.

You can run the program on the text file by typing the following command into the terminal:

```
$ python job.py text.txt
```

Example: unique words

Here is a job that produces **a list of unique words** in a file (i.e. it removes duplicate words). You can run the job on text.txt in the usual way:

```
$ python job.py text.txt
```

Try it: unique lines

The previous example showed how to write a MRJob program that produces a list of unique words in a file. Try modifying that job into one that produces **a list of unique lines** instead. For example, if an input file is this:

```
2013-11-01 aa
2013-11-02 bb
2013-11-03 cc
2013-11-01 aa
2013-11-03 dd
2013-11-02 bb
```

Your job will output these contents (the duplicate rows have been removed, ignored the order and quotation marks):

```
2013-11-01 aa
2013-11-02 bb
2013-11-03 cc
2013-11-03 dd
```

You can run your job using the following command:

```
$ python job.py text.txt
```

Try it: first letter frequency

Try writing a job that counts **the frequencies of word first letters** in a file. That is, for each letter of the alphabet (a-z) it counts the number of words that start with that letter. Let's ignore letter case, so that 'a' and 'A' count as the same letter.

A file "job.py" has been started for you. You just need to fill in the details.

The file text.txt is also on the server. You can run your job using text.txt as the input using the following command:

```
$ python job.py text.txt
```

Try it: average word length

Try writing a job that calculates **the average length of words** in a file.

A file "job.py" has been started for you. You just need to fill in the details.

The file text.txt is also on the server. You can run your job using text.txt as the input using the following command:

```
$ python job.py text.txt
```

Example: using CSV data

Now let's try working with a CSV file.

When you click the panel on the right you will get a terminal connection to a server that has Hadoop and YARN installed.

In your home directory on this server is a CSV file called "employees.csv". You can open the file to inspect its contents. The fields in the file are as follows:

```
employee_id (integer)
first_name (string)
last_name (string)
email (string)
phone_number (string)
hire_date (date)
salary (integer)
```

Suppose we want to know **the maximum salary of employees**. Let's write a MRJob program to calculate it.

Without a combiner

The mapper is applied to only part of the file, so it cannot calculate the overall maximum salary. But it can return each of the salaries, and then let the reducer find the maximum of salaries.

The program "job1.py" does this. You can run the program using employees.csv as the input using the following command:

```
$ python job1.py employees.csv
```

With a combiner

You may have noticed that mapper() is a bit lazy - it doesn't calculate the maximum salary among the salaries that it has, it just returns each salary paired with 1, and lets reducer() do the rest.

We can do better, by adding a combiner, which first finds the local maximum salary before the results get sent to the reducer.

The program "job2.py" does this. You can run the program using employees.csv as the input using the following command:

```
$ python job2.py employees.csv
```

Try it: using CSV data

Consider the same CSV file as last slide. Suppose you want **a list of distinct salaries and, for each of those salaries, the last names of employees on that salary**. Try writing a MRJob job that will do that for you.

When you click the panel on the right you will get a terminal connection to a server that has Hadoop and YARN installed and running, and has the file "employees.csv" in your home directory in the local file system. Here are the fields in the file again:

```
employee_id (integer)
first_name (string)
last_name (string)
email (string)
phone_number (string)
hire_date (date)
salary (integer)
```

A file job.py has been started for you. Your task is to finish writing that file.

You can test your job by running the following command:

```
$ python job.py employees.csv
```

Example: using JSON data

Now let's use MRJob to work with JSON data.

When you click the panel on the right you'll get a connection to a server that has a JSON file in your home directory, called "data.json". It contains data about which students are enrolled in which courses. You can open the file to inspect its contents.

Let's use MRJob to produce **a list of courses and their number of students**. The results should be something like this:

```
ZZEN9021: 1
ZZEN9311: 3
ZZEN9313: 4
ZZSC5806: 4
ZZSC5905: 4
ZZSC9001: 2
```

In the panel on the right is a MRJob program to do this. Note that we have imported the `json` library, and used the `json.loads()` function to load the JSON data into a variable called "people".

You can run the job and see the results by using the following command:

```
$ python job.py data.json
```

Try it: using JSON data

Now try for yourself using MRJob to work with JSON data.

When you click the panel on the right you'll get a connection to a server that has the same JSON file as the last slide in your home directory, "data.json". Again, you can open the file to inspect the data.

First, try writing a MRJob program that **counts the total number of enrolments**. So the result should be this:

```
Total enrolments: 18
```

A file called "job1.py" has been created for you, in which to write your program. You can test it using the following command:

```
$ python job1.py data.json
```

Second, try writing a MRJob program that **lists the students who are in each course**. So the results should be these:

```
ZZEN9021: Jamie
ZZEN9311: Natalie, Sam, Tom
ZZEN9313: Jamie, Natalie, Sam, Tom
ZZSC5806: Natalie, Sam, Sarah, Tom
ZZSC5905: Natalie, Sam, Sarah, Tom
ZZSC9001: Sarah, Tom
```

A file called "job2.py" has been created for you, in which to write your program. You can test it using the following command:

```
$ python job2.py data.json
```

Further resources

The MRJob official website is a good source of further information about MRJob:

The MRJob official website

One more example of MRJob: MapReduce example Break down movie ratings by rating score