mrjob Documentation

Release 0.4.3-dev

Steve Johnson

July 30, 2014

Contents

mrjob lets you write MapReduce jobs in Python 2.5+ and run them on several platforms. You can:

- Write multi-step MapReduce jobs in pure Python
- Test on your local machine
- Run on a Hadoop cluster
- Run in the cloud using Amazon Elastic MapReduce (EMR)

mrjob is licensed under the Apache License, Version 2.0.

To get started, install with `pip`:

```
pip install mrjob
```

and begin reading the tutorial below.

Note: This documentation is for 0.4.3-dev, which is currently in development. Documentation for the stable version of mrjob is hosted at http://pythonhosted.org/mrjob.

CHAPTER 1

Guides

1.1 Why mrjob?

1.1.1 Overview

mrjob is the easiest route to writing Python programs that run on Hadoop. If you use mrjob, you'll be able to test your code locally without installing Hadoop or run it on a cluster of your choice.

Additionally, mrjob has extensive integration with Amazon Elastic MapReduce. Once you're set up, it's as easy to run your job in the cloud as it is to run it on your laptop.

Here are a number of features of mrjob that make writing MapReduce jobs easier:

- Keep all MapReduce code for one job in a single class
- Easily upload and install code and data dependencies at runtime
- Switch input and output formats with a single line of code
- Automatically download and parse error logs for Python tracebacks
- Put command line filters before or after your Python code

If you don't want to be a Hadoop expert but need the computing power of MapReduce, mrjob might be just the thing for you.

1.1.2 Why use mrjob instead of X?

Where X is any other library that helps Hadoop and Python interface with each other.

1. mrjob has more documentation than any other framework or library we are aware of. If you're reading this, it's probably your first contact with the library, which means you are in a great position to provide valuable feedback about our documentation. Let us know if anything is unclear or hard to understand.
2. mrjob lets you run your code without Hadoop at all. Other frameworks require a Hadoop instance to function at all. If you use mrjob, you'll be able to write proper tests for your MapReduce code.
3. mrjob provides a consistent interface across every environment it supports. No matter whether you're running locally, in the cloud, or on your own cluster, your Python code doesn't change at all.
4. mrjob handles much of the machinery of getting code and data to and from the cluster your job runs on. You don't need a series of scripts to install dependencies or upload files.

5. mrjob makes debugging much easier. Locally, it can run a simple MapReduce implementation in-process, so you get a traceback in your console instead of in an obscure log file. On a cluster or on Elastic MapReduce, it parses error logs for Python tracebacks and other likely causes of failure.
6. mrjob automatically serializes and deserializes data going into and coming out of each task so you don't need to constantly `json.loads()` and `json.dumps()`.

1.1.3 Why use X instead of mrjob?

The flip side to mrjob's ease of use is that it doesn't give you the same level of access to Hadoop APIs that Dumbo and Pydoop do. It's simplified a great deal. But that hasn't stopped several companies, including Yelp, from using it for day-to-day heavy lifting. For common (and many uncommon) cases, the abstractions help rather than hinder.

Other libraries can be faster if you use typedbytes. There have been several attempts at integrating it with mrjob, and it may land eventually, but it doesn't exist yet.

1.2 Fundamentals

1.2.1 Installation

Install with `pip`:

```
pip install mrjob
```

or from a git clone of the source code:

```
python setup.py test && python setup.py install
```

1.2.2 Writing your first job

Open a file called `word_count.py` and type this into it:

```
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()
```

Now go back to the command line, find your favorite body of text (such mrjob's `README.rst`, or even your new file `word_count.py`), and try this:

```
$ python word_count.py my_file.txt
```

You should see something like this:

```
"chars" 3654
"lines" 123
"words" 417
```

Congratulations! You've just written and run your first program with mrjob.

What's happening

A job is defined by a class that inherits from `MRJob`. This class contains methods that define the *steps* of your job.

A "step" consists of a mapper, a combiner, and a reducer. All of those are optional, though you must have at least one. So you could have a step that's just a mapper, or just a combiner and a reducer.

When you only have one step, all you have to do is write methods called `mapper()`, `combiner()`, and `reducer()`.

The `mapper()` method takes a key and a value as args (in this case, the key is ignored and a single line of text input is the value) and yields as many key-value pairs as it likes. The `reduce()` method takes a key and an iterator of values and also yields as many key-value pairs as it likes. (In this case, it sums the values for each key, which represent the numbers of characters, words, and lines in the input.)

> **Warning:** Forgetting the following information will result in confusion.

The final required component of a job file is these two lines at the end of the file, **every time**:

```
if __name__ == '__main__':
    MRWordCounter.run()  # where MRWordCounter is your job class
```

These lines pass control over the command line arguments and execution to mrjob. **Without them, your job will not work.** For more information, see *Hadoop Streaming and mrjob* and *Why can't I put the job class and run code in the same file?*.

1.2.3 Running your job different ways

The most basic way to run your job is on the command line:

```
$ python my_job.py input.txt
```

By default, output will be written to stdout.

You can pass input via stdin, but be aware that mrjob will just dump it to a file first:

```
$ python my_job.py < input.txt
```

You can pass multiple input files, mixed with stdin (using the - character):

```
$ python my_job.py input1.txt input2.txt - < input3.txt
```

By default, mrjob will run your job in a single Python process. This provides the friendliest debugging experience, but it's not exactly distributed computing!

You change the way the job is run with the `-r`/`--runner` option. You can use `-r inline` (the default), `-r local`, `-r hadoop`, or `-r emr`.

To run your job in multiple subprocesses with a few Hadoop features simulated, use `-r local`.

To run it on your Hadoop cluster, use `-r hadoop`.

If you have Elastic MapReduce configured (see *Elastic MapReduce Quickstart*), you can run it there with `-r emr`.

Your input files can come from HDFS if you're using Hadoop, or S3 if you're using EMR:

```
$ python my_job.py -r emr s3://my-inputs/input.txt
$ python my_job.py -r hadoop hdfs://my_home/input.txt
```

If your code spans multiple files, see *Putting your source tree in PYTHONPATH*.

1.2.4 Writing your second job

Most of the time, you'll need more than one step in your job. To define multiple steps, override `steps()` and return a list of `mrjob.step.MRStep`.

Here's a job that finds the most commonly used word in the input:

```
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()
```

1.2.5 Configuration

mrjob has an overflowing cornucopia of configuration options. You'll want to specify some on the command line, some in a config file.

You can put a config file at `/etc/mrjob.conf`, `~/.mrjob.conf`, or `./mrjob.conf` for mrjob to find it without passing it via `--conf-path`.

Config files are interpreted as YAML if you have the `yaml` module installed. Otherwise, they are interpreted as JSON.

See *Config file format and location* for in-depth information. Here is an example file:

```
runners:
  emr:
    aws-region: us-west-1
    python_archives:
      - a_library_I_use_on_emr.tar.gz
  inline:
    base_tmp_dir: $HOME/.tmp
```

1.3 Concepts

1.3.1 MapReduce and Apache Hadoop

This section uses text from Apache's MapReduce Tutorial.

MapReduce is a way of writing programs designed for processing vast amounts of data, and a system for running those programs in a distributed and fault-tolerant way. Apache Hadoop is one such system designed primarily to run Java code.

A MapReduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically both the input and the output of the job are stored in a file system shared by all processing nodes. The framework takes care of scheduling tasks, monitoring them, and re-executing the failed tasks.

The MapReduce framework consists of a single master "job tracker" and one slave "task tracker" per cluster-node. The master is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

As the job author, you write *map*, *combine*, and *reduce* functions that are submitted to the job tracker for execution.

A *mapper* takes a single key and value as input, and returns zero or more (key, value) pairs. The pairs from all map outputs of a single step are grouped by key.

A *combiner* takes a key and a subset of the values for that key as input and returns zero or more (key, value) pairs. Combiners are optimizations that run immediately after each mapper and can be used to decrease total data transfer. Combiners should be idempotent (produce the same output if run multiple times in the job pipeline).

A *reducer* takes a key and the complete set of values for that key in the current step, and returns zero or more arbitrary (key, value) pairs as output.

After the reducer has run, if there are more steps, the individual results are arbitrarily assigned to mappers for further processing. If there are no more steps, the results are sorted and made available for reading.

An example

Consider a program that counts how many times words occur in a document. Here is some input:

```
The wheels on the bus go round and round,
round and round, round and round
The wheels on the bus go round and round,
all through the town.
```

The inputs to the mapper will be (`None`, "*one line of text*"). (The key is `None` because the input is just raw text.)

The mapper converts the line to lowercase, removes punctuation, splits it on whitespace, and outputs `(word, 1)` for each item.

```
mapper input: (None, "The wheels on the bus go round and round,")
mapper output:
    "the", 1
    "wheels", 1
    "on", 1
    "the", 1
    "bus", 1
    "go", 1
    "round", 1
    "and", 1
    "round", 1
```

Each call to the combiner gets a word as the key and a list of 1s as the value. It sums the 1s and outputs the original key and the sum.

```
combiner input: ("the", [1, 1])
combiner output:
    "the", 2
```

The reducer is identical to the combiner; for each key, it simply outputs the original key and the sum of the values.

```
reducer input: ("round", [2, 4, 2])
reducer output:
    "round", 8
```

The final output is collected:

```
"all", 1
"and", 4
"bus", 2
"go", 2
"on", 2
"round", 8
"the", 5
"through", 1
"town", 1
"wheels", 2
```

Your algorithm may require several repetitions of this process.

1.3.2 Hadoop Streaming and mrjob

Note: If this is your first exposure to MapReduce or Hadoop, you may want to skip this section and come back later. Feel free to stick with it if you feel adventurous.

Although Hadoop is primarly designed to work with Java code, it supports other languages via *Hadoop Streaming*. This jar opens a subprocess to your code, sends it input via stdin, and gathers results via stdout.

In most cases, the input to a Hadoop Streaming job is a set of newline-delimited files. Each line of input is passed to your mapper, which outputs key-value pairs expressed as two strings separated by a tab and ending with a newline, like this:

```
key1\tvalue1\nkey2\nvalue2\n
```

Hadoop then sorts the output lines by key (the line up to the first tab character) and passes the sorted lines to the appropriate combiners or reducers.

mrjob is a framework that assists you in submitting your job to the Hadoop job tracker and in running each individual step under Hadoop Streaming.

How your program is run

Depending on the way your script is invoked on the command line, it will behave in different ways. You'll only ever use one of these; the rest are for mrjob and Hadoop Streaming to use.

When you run with no arguments or with `--runner`, you invoke mrjob's machinery for running your job or submitting it to the cluster. We'll call it *Process 1* to disambiguate it later. Your mappers and reducers are not called in this process at all [1].

```
$ python my_job.py -r hadoop input.txt  # run process 1
```

Within Process 1, mrjob will need to determine what the *steps* of your project are. It does so by launching another subprocess of your job, this time with the `--steps` argument, which we'll call Process 2:

```
$ python my_job.py --steps              # run process 2
[{"mapper": {"type": "script"},
  "reducer": {"type": "script"},
  "combiner": {"type": "script"},
  "type": "streaming"}]
```

mrjob now has all the information it needs to send the job to Hadoop [2]. It does so. (At this time, please wave your hands in the air to represent magic.)

mrjob has told Hadoop something like this:

- Run a step with Hadoop Streaming.
- The command for the mapper is `python my_job.py --step-num=0 --mapper`.
- The command for the combiner is `python my_job.py --step-num=0 --combiner`.
- The command for the reducer is `python my_job.py --step-num=0 --reducer`.

When Hadoop distributes tasks among the task nodes, Hadoop Streaming will use the appropriate command to process the data it is given. (We did not assign numbers to the above commands because there might be anywhere from 1 to 10,000 processes running on Hadoop.)

You should now have a pretty good idea of the different environments in which your job is run.

1.4 Writing jobs

This guide covers everything you need to know to write your job. You'll probably need to flip between this guide and *Runners* to find all the information you need.

[1] Unless you're using the `inline` runner, which is a special case for debugging.
[2] Or run the job locally.

1.4.1 Defining steps

Your job will be defined in a file to be executed on your machine as a Python script, as well as on a Hadoop cluster as an individual map, combine, or reduce task. (See *How your program is run* for more on that.)

All dependencies must either be contained within the file, available on the task nodes, or uploaded to the cluster by mrjob when your job is submitted. (*Runners* explains how to do those things.)

The following two sections are more reference-oriented versions of *Writing your first job* and *Writing your second job*.

Single-step jobs

The simplest way to write a one-step job is to subclass `MRJob` and override a few methods:

```
from mrjob.job import MRJob
import re

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


class MRWordFreqCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield word.lower(), 1

    def combiner(self, word, counts):
        yield word, sum(counts)

    def reducer(self, word, counts):
        yield word, sum(counts)


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

(See *Writing your first job* for an explanation of this example.)

Here are all the methods you can override to write a one-step job. We'll explain them over the course of this document.

- `mapper()`
- `combiner()`
- `reducer()`
- `mapper_init()`
- `combiner_init()`
- `reducer_init()`
- `mapper_final()`
- `combiner_final()`
- `reducer_final()`
- `mapper_cmd()`
- `combiner_cmd()`

- `reducer_cmd()`
- `mapper_pre_filter()`
- `combiner_pre_filter()`
- `reducer_pre_filter()`

Multi-step jobs

To define multiple steps, override the `steps()` method to return a list of `mr()` calls:

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

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


class MRMostUsedWord(MRJob):

    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):
        # 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)

    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)
        ]


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

(This example is explained further in *Protocols*.)

The keyword arguments accepted by `mr()` are the same as the *method names listed in the previous section*, plus a `jobconf` argument which takes a dictionary of jobconf arguments to pass to Hadoop.

Note: If this is your first time learning about mrjob, you should skip down to *Protocols* and finish this section later.

Setup and teardown of tasks

Remember from *How your program is run* that your script is invoked once per task by Hadoop Streaming. It starts your script, feeds it stdin, reads its stdout, and closes it. mrjob lets you write methods to run at the beginning and end of this process: the `*_init()` and `*_final()` methods:

- `mapper_init()`
- `combiner_init()`
- `reducer_init()`
- `mapper_final()`
- `combiner_final()`
- `reducer_final()`

(And the corresponding keyword arguments to `mr()`.)

If you need to load some kind of support file, like a `sqlite3` database, or perhaps create a temporary file, you can use these methods to do so. (See *File options* for an example.)

`*_init()` and `*_final()` methods can yield values just like normal tasks. Here is our word frequency count example rewritten to use these methods:

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

class MRWordFreqCount(MRJob):

    def init_get_words(self):
        self.words = {}

    def get_words(self, _, line):
        for word in WORD_RE.findall(line):
            word = word.lower()
            self.words.setdefault(word, 0)
            self.words[word] = self.words[word] + 1

    def final_get_words(self):
        for word, val in self.words.iteritems():
            yield word, val

    def sum_words(self, word, counts):
        yield word, sum(counts)

    def steps(self):
        return [MRStep(mapper_init=self.init_get_words,
                       mapper=self.get_words,
                       mapper_final=self.final_get_words,
                       combiner=self.sum_words,
                       reducer=self.sum_words)]
```

In this version, instead of yielding one line per word, the mapper keeps an internal count of word occurrences across all lines this mapper has seen so far. The mapper itself yields nothing. When Hadoop Streaming stops sending data to the map task, mrjob calls `final_get_words()`. That function emits the totals for this task, which is a much smaller set of output lines than the mapper would have output.

The optimization above is similar to using *combiners*, demonstrated in *Multi-step jobs*. It is usually clearer to use a combiner rather than a custom data structure, and Hadoop may run combiners in more places than just the ends of tasks.

Defining command line options has a partial example that shows how to load a `sqlite3` database using `mapper_init()`.

Shell commands as steps

You can forego scripts entirely for a step by specifying it as shell a command. To do so, use `mapper_cmd`, `combiner_cmd`, or `reducer_cmd` as arguments to `MRStep`, or override the methods of the same names on `MRJob`. (See `mapper_cmd()`, `combiner_cmd()`, and `reducer_cmd()`.)

Warning: The default `inline` runner does not support `*_cmd()`. If you want to test locally, use the `local` runner (`-r local`).

You may mix command and script steps at will. This job will count the number of lines containing the string "kitty":

```
from mrjob.job import job


class KittyJob(MRJob):

    OUTPUT_PROTOCOL = JSONValueProtocol

    def mapper_cmd(self):
        return "grep kitty"

    def reducer(self, key, values):
        yield None, sum(1 for _ in values)


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

Step commands are run without a shell. But if you'd like to use shell features such as pipes, you can use `mrjob.util.bash_wrap()` to wrap your command in a call to `bash`.

```
from mrjob.util import bash_wrap

class DemoJob(MRJob):

    def mapper_cmd(self):
        return bash_wrap("grep 'blah blah' | wc -l")
```

Note: You may not use `*_cmd()` with any other options for a task such as `*_filter()`, `*_init()`, `*_final()`, or a regular mapper/combiner/reducer function.

Note: You might see an opportunity here to write your MapReduce code in whatever language you please. If that appeals to you, check out *upload_files* for another piece of the puzzle.

Filtering task input with shell commands

You can specify a command to filter a task's input before it reaches your task using the `mapper_pre_filter` and `reducer_pre_filter` arguments to `MRStep`, or override the methods of the same names on `MRJob`. Doing so will cause mrjob to pipe input through that comand before it reaches your mapper.

> **Warning:** The default `inline` runner does not support `*_pre_filter()`. If you want to test locally, use the `local` runner (`-r local`).

Here's a job that tests filters using **grep**:

```
from mrjob.job import MRJob
from mrjob.protocol import JSONValueProtocol
from mrjob.step import MRStep


class KittiesJob(MRJob):

    OUTPUT_PROTOCOL = JSONValueProtocol

    def test_for_kitty(self, _, value):
        yield None, 0  # make sure we have some output
        if 'kitty' not in value:
            yield None, 1

    def sum_missing_kitties(self, _, values):
        yield None, sum(values)

    def steps(self):
        return [
            MRStep(mapper_pre_filter='grep "kitty"',
                   mapper=self.test_for_kitty,
                   reducer=self.sum_missing_kitties)]


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

The output of the job should always be `0`, since every line that gets to `test_for_kitty()` is filtered by **grep** to have "kitty" in it.

Filter commands are run without a shell. But if you'd like to use shell features such as pipes, you can use `mrjob.util.bash_wrap()` to wrap your command in a call to `bash`. See *Filtering task input with shell commands* for an example of `mrjob.util.bash_wrap()`.

1.4.2 Protocols

mrjob assumes that all data is newline-delimited bytes. It automatically serializes and deserializes these bytes using *protocols*. Each job has an *input protocol*, an *output protocol*, and an *internal protocol*.

A protocol has a `read()` method and a `write()` method. The `read()` method converts bytes to pairs of Python objects representing the keys and values. The `write()` method converts a pair of Python objects back to bytes.

The *input protocol* is used to read the bytes sent to the first mapper (or reducer, if your first step doesn't use a mapper). The *output protocol* is used to write the output of the last step to bytes written to the output file. The *internal protocol* converts the output of one step to the input of the next if the job has more than one step.

You can specify which protocols your job uses like this:

```
class MyMRJob(mrjob.job.MRJob):

    # these are the defaults
    INPUT_PROTOCOL = mrjob.protocol.RawValueProtocol
    INTERNAL_PROTOCOL = mrjob.protocol.JSONProtocol
    OUTPUT_PROTOCOL = mrjob.protocol.JSONProtocol
```

The default input protocol is `RawValueProtocol`, which reads and writes lines of raw text with no key. So by default, the first step in your job sees `(None, <text of the line>)` for each line of input.

The default output and internal protocols are both `JSONProtocol`, which reads and writes JSON strings separated by a tab character. (Hadoop Streaming uses the tab character to separate keys and values within one line when it sorts your data [3].)

If your head hurts a bit, think of it this way: use `RawValueProtocol` when you want to read or write lines of raw text. Use `JSONProtocol` when you want to read or write key-value pairs where the key and value are JSON-enoded bytes.

Note: Hadoop Streaming does not understand JSON, or mrjob protocols. It simply groups lines by doing a string comparison on the keys.

Here are all the protocols mrjob includes:

- `JSONProtocol` / `JSONValueProtocol`: JSON
- `PickleProtocol` / `PickleValueProtocol`: pickle
- `RawProtocol` / `RawValueProtocol`: raw string
- `ReprProtocol` / `ReprValueProtocol`: serialize with `repr()`, deserialize with `mrjob.util.safeeval()`

The `*ValueProtocol` protocols assume the input lines don't have keys, and don't write a key as output.

Data flow walkthrough by example

Let's revisit our example from *Multi-step jobs*. It has two steps and takes a plain text file as input.

```
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)
        ]
```

The first step starts with `mapper_get_words()`:

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

Since the input protocol is `RawValueProtocol`, the key will always be `None` and the value will be the text of the line.

[3] This behavior is configurable, but there is currently no mrjob-specific documentation. GitHub pull requests are always appreciated.

The function discards the key and yields `(word, 1)` for each word in the line. Since the internal protocol is `JSONProtocol`, each component of the output is serialized to JSON. The serialized components are written to stdout separated by a tab character and ending in a newline character, like this:

```
"mrjob" 1
"is"    1
"a" 1
"python"    1
```

The next two parts of the step are the combiner and reducer:

```
def combiner_count_words(self, word, counts):
    # 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)
```

In both cases, bytes are deserialized into `(word, counts)` by `JSONProtocol`, and the output is serialized as JSON in the same way (because both are followed by another step). It looks just like the first mapper output, but the results are summed:

```
"mrjob" 31
"is"    2
"a" 2
"Python"    1
```

The final step is just a reducer:

```
# 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)
```

Since all input to this step has the same key (`None`), a single task will get all rows. Again, `JSONProtocol` will handle deserialization and produce the arguments to `reducer_find_max_word()`.

The output protocol is also `JSONProtocol`, so the final output will be:

```
31  "mrjob"
```

And we're done! But that's a bit ugly; there's no need to write the key out at all. Let's use `JSONValueProtocol` instead, so we only see the JSON-encoded value:

```
class MRMostUsedWord(MRJob):

    OUTPUT_PROTOCOL = JSONValueProtocol
```

Now we should have code that is identical to `examples/mr_most_used_word.py` in mrjob's source code. Let's try running it (`-q` prevents debug logging):

```
$ python mr_most_used_word.py README.txt -q
"mrjob"
```

Hooray!

Specifying protocols for your job

Usually, you'll just want to set one or more of the class variables `INPUT_PROTOCOL`, `INTERNAL_PROTOCOL`, and `OUTPUT_PROTOCOL`:

```
class BasicProtocolJob(MRJob):

    # get input as raw strings
    INPUT_PROTOCOL = RawValueProtocol
    # pass data internally with pickle
    INTERNAL_PROTOCOL = PickleProtocol
    # write output as JSON
    OUTPUT_PROTOCOL = JSONProtocol
```

If you need more complex behavior, you can override `input_protocol()`, `internal_protocol()`, or `output_protocol()` and return a protocol object instance. Here's an example that sneaks a peek at *Defining command line options*:

```
class CommandLineProtocolJob(MRJob):

    def configure_options(self):
        super(CommandLineProtocolJob, self).configure_options()
        self.add_passthrough_option(
            '--output-format', default='raw', choices=['raw', 'json'],
            help="Specify the output format of the job")

    def output_protocol(self):
        if self.options.output_format == 'json':
            return JSONValueProtocol()
        elif self.options.output_format == 'raw':
            return RawValueProtocol()
```

Finally, if you need to use a completely different concept of protocol assignment, you can override `pick_protocols()`:

```
class WhatIsThisIDontEvenProtocolJob(MRJob):

    def pick_protocols(self, step_num, step_type):
        return random.choice([Protocololol, ROFLcol, Trolltocol, Locotorp])
```

Writing custom protocols

A protocol is an object with methods `read(self, line)` and `write(self, key, value)`. The `read()` method takes a string and returns a 2-tuple of decoded objects, and `write()` takes the key and value and returns the line to be passed back to Hadoop Streaming or as output.

Here is a simplified version of mrjob's JSON protocol:

```
import json


class JSONProtocol(object):

    def read(self, line):
        k_str, v_str = line.split('\t', 1)
        return json.loads(k_str), json.loads(v_str)
```

```
    def write(self, key, value):
        return '%s\t%s' % (json.dumps(key), json.dumps(value))
```

You can improve performance significantly by caching the serialization/deserialization results of keys. Look at the source code of `mrjob.protocol` for an example.

Jar steps

You can run Java directly on Hadoop (bypassing Hadoop Streaming) by using `JarStep` instead of `MRStep()`.

For example, on EMR you can use a jar to run a script:

```
from mrjob.job import MRJob
from mrjob.step import JarStep

class ScriptyJarJob(MRJob):

    def steps(self):
        return [JarStep(
            jar='s3://elasticmapreduce/libs/script-runner/script-runner.jar',
            args=['s3://my_bucket/my_script.sh'])]
```

More interesting is combining `MRStep` and `JarStep` in the same job. Use `JarStep.INPUT` and `JarStep.OUTPUT` in *args* to stand for the input and output paths for that step. For example:

```
class NaiveBayesJob(MRJob):

    def steps(self):
        return [
            MRStep(mapper=self.mapper, reducer=self.reducer),
            JarStep(
                jar='elephant-driver.jar',
                args=['naive-bayes', JarStep.INPUT, JarStep.OUTPUT]
            )
        ]
```

`JarStep` has no concept of *Protocols*. If your jar reads input from a `MRStep`, or writes input read by another `MRStep`, it is up to those steps to read and write data in the format your jar expects.

If you are writing the jar yourself, the easiest solution is to have it read and write mrjob's default protocol (lines containing two JSONs, separated by a tab).

If you are using a third-party jar, you can set custom protocols for the steps before and after it by overriding `pick_protocols()`.

> **Warning:** If the first step of your job is a `JarStep` and you pass in multiple input paths, mrjob will replace `JarStep.INPUT` with the input paths joined together with a comma. Not all jars can handle this!
> Best practice in this case is to put all your input into a single directory and pass that as your input path.

1.4.3 Defining command line options

Recall from *How your program is run* that your script is executed in several contexts: once for the initial invokation, and once for each task. If you just add an option to your job's option parser, that option's value won't be propagated to other runs of your script. Instead, you can use mrjob's option API: `add_passthrough_option()` and `add_file_option()`.

Passthrough options

A *passthrough option* is an `optparse` option that mrjob is aware of. mrjob inspects the value of the option when you invoke your script [4] and reproduces that value when it invokes your script in other contexts. The command line-switchable protocol example from before uses this feature:

```
class CommandLineProtocolJob(MRJob):

    def configure_options(self):
        super(CommandLineProtocolJob, self).configure_options()
        self.add_passthrough_option(
            '--output-format', default='raw', choices=['raw', 'json'],
            help="Specify the output format of the job")

    def output_protocol(self):
        if self.options.output_format == 'json':
            return JSONValueProtocol()
        elif self.options.output_format == 'raw':
            return RawValueProtocol()
```

When you run your script with `--output-format=json`, mrjob detects that you passed `--output-format` on the command line. When your script is run in any other context, such as on Hadoop, it adds `--output-format=json` to its command string.

`add_passthrough_option()` takes the same arguments as `optparse.OptionParser.add_option()`. For more information, see the optparse docs.

File options

A *file option* is like a passthrough option, but:

1. Its value must be a string or list of strings (`action="store"` or `action="append"`), where each string represents either a local path, or an HDFS or S3 path that will be accessible from the task nodes.
2. That file will be downloaded to each task's local directory and the value of the option will magically be changed to its path.

For example, if you had a map task that required a `sqlite3` database, you could do this:

```
class SqliteJob(MRJob):

    def configure_options(self):
        super(SqliteJob, self).configure_options()
        self.add_file_option('--database')

    def mapper_init(self):
        # make sqlite3 database available to mapper
        self.sqlite_conn = sqlite3.connect(self.options.database)
```

You could call it any of these ways, depending on where the file is:

```
$ python sqlite_job.py -r local  --database=/etc/my_db.sqlite3
$ python sqlite_job.py -r hadoop --database=/etc/my_db.sqlite3
$ python sqlite_job.py -r hadoop --database=hdfs://my_dir/my_db.sqlite3
$ python sqlite_job.py -r emr    --database=/etc/my_db.sqlite3
$ python sqlite_job.py -r emr    --database=s3://my_bucket/my_db.sqlite3
```

[4] This is accomplished using crazy `optparse` hacks so you don't need to limit yourself to certain option types. However, your default values need to be compatible with `copy.deepcopy()`.

In any of these cases, when your task runs, `my_db.sqlite3` will always be available in the task's working directory, and the value of `self.options.database` will always be set to its path.

See *Making files available to tasks* if you want to upload a file to your tasks' working directories without writing a custom command line option.

> **Warning:** You **must** wait to read files until **after class initialization**. That means you should use the **_init()* methods to read files. Trying to read files into class variables will not work.

Custom option types

`optparse` allows you to add custom types and actions to your options (see Extending optparse), but doing so requires passing a custom `Option` object into the `OptionParser` constructor. mrjob creates its own `OptionParser` object, so if you want to use a custom `Option` class, you'll need to set the `OPTION_CLASS` attribute.

```
import optparse

import mrjob


class MyOption(optparse.Option):
    pass    # extend optparse as documented by the Python standard library


class MyJob(mrjob.job.MRJob):

    OPTION_CLASS = MyOption
```

1.4.4 Counters

Hadoop lets you track *counters* that are aggregated over a step. A counter has a group, a name, and an integer value. Hadoop itself tracks a few counters automatically. mrjob prints your job's counters to the command line when your job finishes, and they are available to the runner object if you invoke it programmatically.

To increment a counter from anywhere in your job, use the `increment_counter()` method:

```
class MRCountingJob(MRJob):

    def mapper(self, _, value):
        self.increment_counter('group', 'counter_name', 1)
        yield _, value
```

At the end of your job, you'll get the counter's total value:

```
group:
    counter_name: 1
```

1.5 Runners

While the `MRJob` class is the part of the framework that handles the execution of your code in a MapReduce context, the **runner** is the part that packages and submits your job to be run, and reporting the results back to you.

In most cases, you will interact with runners via the command line and configuration files. When you invoke mrjob via the command line, it reads your command line options (the `--runner` parameter) to determine which type of runner

to create. Then it creates the runner, which reads your configuration files and command line args and starts your job running in whatever context you chose.

Most of the time, you won't have any reason to construct a runner directly. Instead you'll invoke your Python script on the command line and it will make a runner automatically, you'll call `mrjob run my_script` to have the `mrjob` command build a runner for your script (which may or may not be Python), or you'll write some sort of wrapper that calls `my_job.make_runner()`.

Internally, the general order of operations is:

- Get a runner by calling `make_runner()` on your job
- Call `run()` on your runner. This will:
  - Run your job with *`--steps`* to find out how many mappers/reducers to run
  - Copy your job and supporting files to Hadoop
  - Instruct Hadoop to run your job with the appropriate *`--mapper`*, *`--combiner`*, *`--reducer`*, and *`--step-num`* arguments

Each runner runs a single job once; if you want to run a job multiple times, make multiple runners.

Subclasses: `EMRJobRunner`, `HadoopJobRunner`, `InlineMRJobRunner`, `LocalMRJobRunner`

1.5.1 Testing locally

To test the job locally, just run:

```
python your_mr_job_sub_class.py < log_file_or_whatever > output
```

The script will automatically invoke itself to run the various steps, using `InlineMRJobRunner` (`--runner=inline`). If you want to simulate Hadoop more closely, you can use `--runner=local`, which doesn't add your working directory to the `PYTHONPATH`, sets a few Hadoop environment variables, and uses multiple subprocesses for tasks.

You can also run individual steps:

```
# test 1st step mapper:
python your_mr_job_sub_class.py --mapper
# test 2nd step reducer (step numbers are 0-indexed):
python your_mr_job_sub_class.py --reducer --step-num=1
```

By default, we read from stdin, but you can also specify one or more input files. It automatically decompresses .gz and .bz2 files:

```
python your_mr_job_sub_class.py log_01.gz log_02.bz2 log_03
```

See `mrjob.examples` for more examples.

1.5.2 Running on your own Hadoop cluster

- Set up a hadoop cluster (see http://hadoop.apache.org/common/docs/current/)
- If running Python 2.5 on your cluster, install the `simplejson` module on all nodes (recommended but not required for Python 2.6+).
- Make sure `HADOOP_HOME` is set
- Run your job with `-r hadoop`:

```
python your_mr_job_sub_class.py -r hadoop < input > output
```

1.5.3 Running on EMR

- Set up your Amazon account and credentials (see *Configuring AWS credentials*)
- Run your job with `-r emr`:

```
python your_mr_job_sub_class.py -r emr < input > output
```

1.5.4 Configuration

Runners are configured by several methods:

- from `mrjob.conf` (see *Config file format and location*)
- from the command line
- by re-defining `job_runner_kwargs()` etc in your `MRJob` (see *Job runner configuration*)
- by instantiating the runner directly

In most cases, you should put all configuration in `mrjob.conf` and use the command line args or class variables to customize how individual jobs are run.

1.5.5 Running your job programmatically

It is fairly common to write an organization-specific wrapper around mrjob. Use `make_runner()` to run an `MRJob` from another Python script. The context manager guarantees that all temporary files are cleaned up regardless of the success or failure of your job.

This pattern can also be used to write integration tests (see *Testing jobs*).

```
from __future__ import with_statement # only needed on Python 2.5

mr_job = MRWordCounter(args=['-r', 'emr'])
with mr_job.make_runner() as runner:
    runner.run()
    for line in runner.stream_output():
        key, value = mr_job.parse_output_line(line)
        ... # do something with the parsed output
```

You the `MRJob`, use a context manager to create the runner, run the job, iterate over the output lines, and use the job instance to parse each line with its output protocol.

Further reference:

- `make_runner()`
- `stream_output()`
- `parse_output_line()`

Limitations

Note: You should pay attention to the next sentence.

You cannot use the programmatic runner functionality in the same file as your job class. As an example of what not to do, here is some code that does not work.

Warning: The code below shows you what **not** to do.

```
from mrjob.job import MRJob

class MyJob(MRJob):
    # (your job)

mr_job = MyJob(args=[args])
with mr_job.make_runner() as runner:
    runner.run()
    # ... etc
```

If you try to do this, mrjob will give you an error message similar or identical to this one:

```
UsageError: make_runner() was called with --steps. This probably means you
            tried to use it from __main__, which doesn't work.
```

What you need to do instead is put your job in one file, and your run code in another. Here are two files that would correctly handle the above case.

```
# job.py
from mrjob.job import MRJob

class MyJob(MRJob):
    # (your job)

if __name__ == '__main__':
    MyJob.run()

# run.py
from job import MyJob
mr_job = MyJob(args=[args])
with mr_job.make_runner() as runner:
    runner.run()
    # ... etc
```

Why can't I put the job class and run code in the same file?

The file with the job class is sent to Hadoop to be run. Therefore, the job file cannot attempt to start the Hadoop job, or you would be recursively creating Hadoop jobs!

The code that runs the job should only run *outside* of the Hadoop context.

The `if __name__ == '__main__'` block is only run if you invoke the job file as a script. It is not run when imported. That's why you can import the job class to be run, but it can still be invoked as an executable.

Counters

Counters may be read through the `counters()` method on the runner. The example below demonstrates the use of counters in a test case.

mr_counting_job.py

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


class MRCountingJob(MRJob):

    def steps(self):
        # 3 steps so we can check behavior of counters for multiple steps
        return [MRStep(self.mapper),
                MRStep(self.mapper),
                MRStep(self.mapper)]

    def mapper(self, _, value):
        self.increment_counter('group', 'counter_name', 1)
        yield _, value


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

test_counters.py

```
from __future__ import with_statement

try:
    import unittest2 as unittest
except ImportError:
    import unittest

from tests.mr_counting_job import MRCountingJob


class CounterTestCase(unittest.TestCase):

    def test_counters(self):
        stdin = StringIO('foo\nbar\n')

        mr_job = MRCountingJob(['--no-conf', '-'])
        mr_job.sandbox(stdin=stdin)

        with mr_job.make_runner() as runner:
            runner.run()

            self.assertEqual(runner.counters(),
                             [{'group': {'counter_name': 2}},
                              {'group': {'counter_name': 2}},
                              {'group': {'counter_name': 2}}])
```

1.6 Config file format and location

We look for `mrjob.conf` in these locations:

- The location specified by `MRJOB_CONF`
- `~/.mrjob.conf`
- `/etc/mrjob.conf`

You can specify one or more configuration files with the *`--conf-path`* flag. See *Options available to all runners* for more information.

The point of `mrjob.conf` is to let you set up things you want every job to have access to so that you don't have to think about it. For example:

- libraries and source code you want to be available for your jobs
- where temp directories and logs should go
- security credentials

`mrjob.conf` is just a YAML- or JSON-encoded dictionary containing default values to pass in to the constructors of the various runner classes. Here's a minimal `mrjob.conf`:

```
runners:
  emr:
    cmdenv:
      TZ: America/Los_Angeles
```

Now whenever you run `mr_your_script.py -r emr`, `EMRJobRunner` will automatically set `TZ` to `America/Los_Angeles` in your job's environment when it runs on EMR.

If you don't have the `yaml` module installed, you can use JSON in your `mrjob.conf` instead (JSON is a subset of YAML, so it'll still work once you install `yaml`). Here's how you'd render the above example in JSON:

```
{
  "runners": {
    "emr": {
      "cmdenv": {
        "TZ": "America/Los_Angeles"
      }
    }
  }
}
```

1.6.1 Precedence and combining options

Options specified on the command-line take precedence over `mrjob.conf`. Usually this means simply overriding the option in `mrjob.conf`. However, we know that *cmdenv* contains environment variables, so we do the right thing. For example, if your `mrjob.conf` contained:

```
runners:
  emr:
    cmdenv:
      PATH: /usr/local/bin
      TZ: America/Los_Angeles
```

and you ran your job as:

```
mr_your_script.py -r emr --cmdenv TZ=Europe/Paris --cmdenv PATH=/usr/sbin
```

We'd automatically handle the `PATH` variables and your job's environment would be:

```
{'TZ': 'Europe/Paris', 'PATH': '/usr/sbin:/usr/local/bin'}
```

What's going on here is that *cmdenv* is associated with `combine_envs()`. Each option is associated with an appropriate combiner function that that combines options in an appropriate way.

Combiner functions can also do useful things like expanding environment variables and globs in paths. For example, you could set:

```
runners:
  local:
    upload_files: &upload_files
    - $DATA_DIR/*.db
  hadoop:
    upload_files: *upload_files
  emr:
    upload_files: *upload_files
```

and every time you ran a job, every job in your `.db` file in `$DATA_DIR` would automatically be loaded into your job's current working directory.

Also, if you specified additional files to upload with *`--file`*, those files would be uploaded in addition to the `.db` files, rather than instead of them.

See *Configuration quick reference* for the entire dizzying array of configurable options.

1.6.2 Option data types

The same option may be specified multiple times and be one of several data types. For example, the AWS region may be specified in `mrjob.conf`, in the arguments to `EMRJobRunner`, and on the command line. These are the rules used to determine what value to use at runtime.

Values specified "later" refer to an option being specified at a higher priority. For example, a value in `mrjob.conf` is specified "earlier" than a value passed on the command line.

When there are multiple values, they are "combined with" a *combiner function*. The combiner function for each data type is listed in its description.

Simple data types

When these are specified more than once, the last non-`None` value is used.

String Simple, unchanged string. Combined with `combine_values()`.

Command String containing all ASCII characters to be parsed with `shlex.split()`, or list of command + arguments. Combined with `combine_cmds()`.

Path Local path with ~ and environment variables (e.g. `$TMPDIR`) resolved. Combined with `combine_paths()`.

List data types

The values of these options are specified as lists. When specified more than once, the lists are concatenated together.

String list List of *strings*. Combined with `combine_lists()`.

Path list List of *paths*. Combined with `combine_path_lists()`.

Dict data types

The values of these options are specified as dictionaries. When specified more than once, each has custom behavior described below.

Plain dict Values specified later override values specified earlier.

Environment variable dict Values specified later override values specified earlier, **except for those with keys ending in ''PATH''**, in which values are concatenated and separated by a colon (`:`) rather than overwritten. The later value comes first.

For example, this config:

```
runners: {emr: {cmdenv: {PATH: "/usr/bin"}}}
```

when run with this command:

```
python my_job.py --cmdenv PATH=/usr/local/bin
```

will result in the following value of `cmdenv`:

```
/usr/local/bin:/usr/bin
```

The one exception to this behavior is in the `local` runner, which uses the local system separator (on Windows `;`, on everything else still `:`) instead of always using `:`.

1.6.3 Using multiple config files

If you have several standard configurations, you may want to have several config files "inherit" from a base config file. For example, you may have one set of AWS credentials, but two code bases and default instance sizes. To accomplish this, use the `include` option:

`~/mrjob.very-large.conf`:

```
include: ~/.mrjob.base.conf
runners:
    emr:
        num_ec2_core_instances: 20
        ec2_core_instance_type: m1.xlarge
```

`~/mrjob.very-small.conf`:

```
include: $HOME/.mrjob.base.conf
runners:
    emr:
        num_ec2_core_instances: 2
        ec2_core_instance_type: m1.small
```

`~/.mrjob.base.conf`:

```
runners:
    emr:
        aws_access_key_id: HADOOPHADOOPBOBADOOP
        aws_region: us-west-1
        aws_secret_access_key: MEMIMOMADOOPBANANAFANAFOFADOOPHADOOP
```

Options that are lists, commands, dictionaries, etc. combine the same way they do between the config files and the command line (with combiner functions).

You can use `$ENVIRONMENT_VARIABLES` and `~/file_in_your_home_dir` inside `include`.

You can inherit from multiple config files by passing `include` a list instead of a string. Files on the right will have precedence over files on the left. To continue the above examples, this config:

`~/.mrjob.everything.conf`

```
include:
- ~/.mrjob.very-small.conf
- ~/.mrjob.very-large.conf
```

will be equivalent to this one:

`~/.mrjob.everything-2.conf`

```
runners:
    emr:
        aws_access_key_id: HADOOPHADOOPBOBADOOP
        aws_region: us-west-1
        aws_secret_access_key: MEMIMOMADOOPBANANAFANAFOFADOOPHADOOP
        num_ec2_core_instances: 20
        ec2_core_instace_type: m1.xlarge
```

In this case, `~/.mrjob.very-large.conf` has taken precedence over `~/.mrjob.very-small.conf`.

1.7 Options available to all runners

The format of each item in this document is:

mrjob_conf_option_name (`--command-line-option-name`) [option_type] **Default:** default value

> Description of option behavior

Options that take multiple values can be passed multiple times on the command line. All options can be passed as keyword arguments to the runner if initialized programmatically.

1.7.1 Making files available to tasks

Most jobs have dependencies of some sort - Python packages, Debian packages, data files, etc. This section covers options available to all runners that mrjob uses to upload files to your job's execution environments. See *File options* if you want to write your own command line options related to file uploading.

> **Warning:** You **must** wait to read files until **after class initialization**. That means you should use the **_init()* methods to read files. Trying to read files into class variables will not work.

bootstrap_mrjob (`--bootstrap-mrjob`, `--no-bootstrap-mrjob`) [boolean] **Default:** `True`

> Should we automatically tar up the mrjob library and install it when we run job? Set this to `False` if you've already installed `mrjob` on your Hadoop cluster or install it by some other method.

upload_files (`--file`) [*path list*] **Default:** `[]`

> Files to copy to the local directory of the mr_job script when it runs. You can set the local name of the dir we unpack into by appending `#localname` to the path; otherwise we just use the name of the file.
>
> In the config file:
>
> ```
> upload_files:
>   - file_1.txt
>   - file_2.sqlite
> ```

On the command line:

```
--file file_1.txt --file file_2.sqlite
```

upload_archives (`--archive`) [*path list*] **Default:** `[]`

A list of archives (e.g. tarballs) to unpack in the local directory of the mr_job script when it runs. You can set the local name of the dir we unpack into by appending `#localname` to the path; otherwise we just use the name of the archive file (e.g. `foo.tar.gz` is unpacked to the directory `foo.tar.gz/`, and `foo.tar.gz#stuff` is unpacked to the directory `stuff/`).

python_archives (`--python-archive`) [*path list*] **Default:** `[]`

Same as upload_archives, except they get added to the job's `PYTHONPATH`.

1.7.2 Temp files and cleanup

base_tmp_dir (`--base-tmp-dir`) [*path*] **Default:** value of `tempfile.gettempdir()`

Path to put local temp dirs inside.

cleanup (`--cleanup`) [*string*] **Default:** `'ALL'`

List of which kinds of directories to delete when a job succeeds. Valid choices are:

- **`'ALL'`: delete local scratch, remote scratch, and logs; stop job flow** if on EMR and the job is not done when cleanup is run.
- `'LOCAL_SCRATCH'`: delete local scratch only
- `'LOGS'`: delete logs only
- `'NONE'`: delete nothing
- `'REMOTE_SCRATCH'`: delete remote scratch only
- `'SCRATCH'`: delete local and remote scratch, but not logs
- `'JOB'`: stop job if on EMR and the job is not done when cleanup runs
- **`'JOB_FLOW'`: terminate the job flow if on EMR and the job is not done** on cleanup
- **`'IF_SUCCESSFUL'` (deprecated): same as `ALL`. Not supported for `cleanup_on_failure`.**

In the config file:

```
cleanup: [LOGS, JOB]
```

On the command line:

```
--cleanup=LOGS,JOB
```

cleanup_on_failure (`--cleanup-on-failure`) [*string*] **Default:** `'NONE'`

Which kinds of directories to clean up when a job fails. Valid choices are the same as **cleanup**.

output_dir (`--output-dir`) [*string*] **Default:** (automatic)

An empty/non-existent directory where Hadoop streaming should put the final output from the job. If you don't specify an output directory, we'll output into a subdirectory of this job's temporary directory. You can control this from the command line with `--output-dir`. This option cannot be set from configuration files. If used with the `hadoop` runner, this path does not need to be fully qualified with `hdfs://` URIs because it's understood that it has to be on HDFS.

no_output (`--no-output`) [boolean] **Default:** `False`

Don't stream output to STDOUT after job completion. This is often used in conjunction with `--output-dir` to store output only in HDFS or S3.

1.7.3 Job execution context

cmdenv (`--cmdenv`) [*environment variable dict*] **Default:** `{}`

Dictionary of environment variables to pass to the job inside Hadoop streaming.

In the config file:

```
cmdenv:
    PYTHONPATH: $HOME/stuff
    TZ: America/Los_Angeles
```

On the command line:

```
--cmdenv PYTHONPATH=$HOME/stuff,TZ=America/Los_Angeles
```

interpreter (`--interpreter`) [*string*] **Default:** value of *python_bin* (`'python'`)

Interpreter to launch your script with. Defaults to the value of **python_bin**, which is deprecated. Change this if you're using a language besides Python 2.5-2.7.

python_bin (`--python-bin`) [*command*] **Default:** `'python'`

Deprecated (use *interpreter* instead). Name/path of alternate Python binary for wrapper scripts and mappers/reducers.

setup (`--setup`) [*string list*] **Default:** `[]`

A list of lines of shell script to run before each task (mapper/reducer).

This option is complex and powerful; the best way to get started is to read the *Job Environment Setup Cookbook*.

Using this option replaces your task with a shell "wrapper" script that executes the setup commands, and then executes the task as the last line of the script. This means that environment variables set by hadoop (e.g. `$mapred_job_id`) are available to setup commands, and that you can pass environment variables to the task (e.g. `$PYTHONPATH`) using `export`.

We use file locking around the setup commands (not the task) to ensure that multiple tasks running on the same node won't run them simultaneously (it's safe to run `make`). Before running the task, we `cd` back to the original working directory.

In addition, passing expressions like `path#name` will cause *path* to be automatically uploaded to the task's working directory with the filename *name*, marked as executable, and interpolated into the script by its absolute path on the machine running the script.

path may also be a URI, and `~` and environment variables within *path* will be resolved based on the local environment. *name* is optional. You can indicate that an archive should be unarchived into a directory by putting a `/` after *name*. For details of parsing, see `parse_setup_cmd()`.

setup_cmds (`--setup_cmd`) [*string list*] **Default:** `[]`

Deprecated since version 0.4.2.

A list of commands to run before each mapper/reducer step. Basically *setup* without automatic file uploading/interpolation. Can also take commands as lists of arguments.

setup_scripts (`--setup-script`) [*path list*] **Default:** `[]`

Deprecated since version 0.4.2.

Files that will be copied into the local working directory and then run.

Pass `'path/to/script#'` to *setup* instead.

sh_bin (`--sh-bin`) [*command*] **Default: sh -e** (**/bin/sh -e** on EMR)

Name/path of alternate shell binary to use for *setup* and *bootstrap*. Needs to be backwards compatible with Bourne Shell (e.g. `'bash'`).

To force setup/bootstrapping to terminate when any command exits with an error, use `'sh -e'`.

steps_python_bin (`--steps-python-bin`) [*command*] **Default:** current Python interpreter

Name/path of alternate python binary to use to query the job about its steps. Rarely needed. Defaults to `sys.executable` (the current Python interpreter).

strict_protocols (`--strict-protocols`, `--no-strict-protocols`) [boolean] **Default:** `None`

If this is true, the job will raise an exception when encountering input or output that can't be handled by its protocols (the default is to increment a counter and continue).

1.7.4 Other

conf_paths (`-c`, `--conf-path`, `--no-conf`) [*path list*] **Default:** see `find_mrjob_conf()`

List of paths to configuration files. This option cannot be used in configuration files, because that would cause a universe-ending causality paradox. Use *–no-conf* on the command line or *conf_paths=[]* to force mrjob to load no configuration files at all. If no config path flags are given, mrjob will look for one in the locations specified in *Config file format and location*.

Config path flags can be used multiple times to combine config files, much like the **include** config file directive. Using *`--no-conf`* will cause mrjob to ignore all preceding config path flags.

For example, this line will cause mrjob to combine settings from `left.conf` and `right .conf`:

```
python my_job.py -c left.conf -c right.conf
```

This line will cause mrjob to read no config file at all:

```
python my_job.py --no-conf
```

This line will cause mrjob to read only `right.conf`, because `--no-conf` nullifies `-c left.conf`:

```
python my_job.py -c left.conf --no-conf -c right.conf
```

1.7.5 Options ignored by the inline runner

These options are ignored because they require a real instance of Hadoop:

- *hadoop_extra_args*
- `hadoop_input_format`
- `hadoop_output_format`,
- *hadoop_streaming_jar*
- *jobconf*

- *partitioner*

These options are ignored because the `inline` runner does not invoke the job as a subprocess or run it in its own directory:

- *cmdenv*
- *python_bin*
- *setup_cmds*
- *setup_scripts*
- *steps_python_bin*
- *upload_archives*
- *upload_files*

1.8 Hadoop-related options

Since mrjob is geared toward Hadoop, there are a few Hadoop-specific options. However, due to the difference between the different runners, the Hadoop platform, and Elastic MapReduce, they are not all available for all runners.

1.8.1 Options available to local, hadoop, and emr runners

These options are both used by Hadoop and simulated by the `local` runner to some degree.

hadoop_version (`--hadoop-version`) [*string*] **Default:** inferred from environment/AWS

Set the version of Hadoop to use on EMR or simulate in the `local` runner. If using EMR, consider setting *ami_version* instead; only AMI version 1.0 supports multiple versions of Hadoop anyway. If *ami_version* is not set, we'll default to Hadoop 0.20 for backwards compatibility with `mrjob` v0.3.0.

jobconf (`--jobconf`) [*dict*] **Default:** `{}`

`-jobconf` args to pass to hadoop streaming. This should be a map from property name to value. Equivalent to passing `['-jobconf', 'KEY1=VALUE1', '-jobconf', 'KEY2=VALUE2', ...]` to *hadoop_extra_args*.

1.8.2 Options available to hadoop and emr runners

hadoop_extra_args (`--hadoop-extra-arg`) [*string list*] **Default:** `[]`

Extra arguments to pass to hadoop streaming. This option is called **extra_args** when passed as a keyword argument to `MRJobRunner`.

hadoop_streaming_jar (`--hadoop-streaming-jar`) [*string*] **Default:** automatic

Path to a custom hadoop streaming jar. This is optional for the `hadoop` runner, which will search for it in `HADOOP_HOME`. The emr runner can take a path either local to your machine or on S3.

label (`--label`) [*string*] **Default:** script's module name, or `no_script`

Description of this job to use as the part of its name.

owner (`--owner`) [*string*] **Default:** `getpass.getuser()`, or `no_user` if that fails

Who is running this job. Used solely to set the job name.

partitioner (`--partitioner`) [*string*] **Default:** `None`

Optional name of a Hadoop partitoner class, e.g. `'org.apache.hadoop.mapred.lib.HashPartitioner'`. Hadoop Streaming will use this to determine how mapper output should be sorted and distributed to reducers. You can also set this option on your job class with the `PARTITIONER` attribute or the `partitioner()` method.

1.8.3 Options available to hadoop runner only

check_input_paths (`--check-input-paths`, `--no-check-input-paths`) [boolean] **Default:** `True`

Option to skip the input path check. With `--no-check-input-paths`, input paths to the runner will be passed straight through, without checking if they exist.

New in version 0.4.1.

hadoop_bin (`--hadoop-bin`) [*command*] **Default:** *hadoop_home* plus `bin/hadoop`

Name/path of your hadoop program (may include arguments).

hadoop_home (`--hadoop-home`) [*path*] **Default:** `HADOOP_HOME`

Alternative to setting the `HADOOP_HOME` environment variable.

hdfs_scratch_dir (`--hdfs-scratch-dir`) [*path*] **Default:** `tmp/`

Scratch space on HDFS. This path does not need to be fully qualified with `hdfs://` URIs because it's understood that it has to be on HDFS.

1.9 Configuration quick reference

1.9.1 Setting configuration options

You can set an option by:

- Passing it on the command line with the switch version (like `--some-option`)
- Passing it as a keyword argument to the runner constructor, if you are creating the runner programmatically
- Putting it in one of the included config files under a runner name, like this:

```
runners:
    local:
        python_bin: python2.6  # only used in local runner
    emr:
        python_bin: python2.5  # only used in Elastic MapReduce runner
```

See *Config file format and location* for information on where to put config files.

1.9.2 Options that can't be set from mrjob.conf (all runners)

For some options, it doesn't make sense to be able to set them in the config file. These can only be specified when calling the constructor of `MRJobRunner`, as command line options, or sometimes by overriding some attribute or method of your `MRJob` subclass.

Runner kwargs or command line

Config	Command line	Default	Type
conf_paths	*-c, –conf-path, –no-conf*	see `find_mrjob_conf()`	*path list*
no_output	*–no-output*	`False`	boolean
output_dir	*–output-dir*	(automatic)	*string*
partitioner	*–partitioner*	`None`	*string*

Runner kwargs or method overrides

Option	Method	Default
`extra_args`	`add_passthrough_option()`	`[]`
`file_upload_args`	`add_file_option()`	`[]`
`hadoop_input_format`	`hadoop_input_format()`	`None`
`hadoop_output_format`	`hadoop_output_format()`	`None`

1.9.3 Other options for all runners

These options can be passed to any runner without an error, though some runners may ignore some options. See the text after the table for specifics.

Config	Command line	Default	Type
base_tmp_dir	*–base-tmp-dir*	value of `tempfile.gettempdir()`	*path*
bootstrap	*–bootstrap*	`[]`	*string list*
bootstrap_mrjob	*–bootstrap-mrjob*, *–no-bootstrap-mrjob*	`True`	boolean
cleanup	*–cleanup*	`'ALL'`	*string*
cleanup_on_failure	*–cleanup-on-failure*	`'NONE'`	*string*
cmdenv	*–cmdenv*	`{}`	*environment variable dict*
hadoop_extra_args	*–hadoop-extra-arg*	`[]`	*string list*
hadoop_streaming_jar	*–hadoop-streaming-jar*	automatic	*string*
interpreter	*–interpreter*	value of *python_bin* (`'python'`)	*string*
jobconf	*–jobconf*	`{}`	*dict*
label	*–label*	script's module name, or `no_script`	*string*
owner	*–owner*	`getpass.getuser()`, or `no_user` if that fails	*string*
python_archives	*–python-archive*	`[]`	*path list*
python_bin	*–python-bin*	`'python'`	*command*
setup	*–setup*	`[]`	*string list*
setup_cmds	*–setup_cmd*	`[]`	*string list*
setup_scripts	*–setup-script*	`[]`	*path list*
sh_bin	*–sh-bin*	**sh -e** (**/bin/sh -e** on EMR)	*command*
steps_python_bin	*–steps-python-bin*	current Python interpreter	*command*
strict_protocols	*–strict-protocols*, *–no-strict-protocols*	`None`	boolean
upload_archives	*–archive*	`[]`	*path list*
upload_files	*–file*	`[]`	*path list*

`LocalMRJobRunner` takes no additional options, but:

- *bootstrap_mrjob* is `False` by default
- *cmdenv* uses the local system path separator instead of `:` all the time (so `;` on Windows, no change elsewhere)
- *python_bin* defaults to the current Python interpreter

In addition, it ignores *hadoop_input_format*, *hadoop_output_format*, *hadoop_streaming_jar*, and *jobconf*

`InlineMRJobRunner` works like `LocalMRJobRunner`, only it also ignores *bootstrap_mrjob*, *cmdenv*, *python_bin*, *setup_cmds*, *setup_scripts*, *steps_python_bin*, *upload_archives*, and *upload_files*.

1.9.4 Additional options for `EMRJobRunner`

Config	Command line	Default	Typ
additional_emr_info	*–additional-emr-info*	`None`	specia
ami_version	*–ami-version*	`'latest'`	*string*
aws_access_key_id	*–aws-access-key-id*	`None`	*string*
aws_availability_zone	*–aws-availability-zone*	AWS default	*string*
aws_region	*–aws-region*	infer from scrach bucket region	*string*
aws_secret_access_key	*–aws-secret-access-key*	`None`	*string*
bootstrap_actions	*–bootstrap-actions*	`[]`	*string*
bootstrap_cmds	*–bootstrap-cmd*	`[]`	*string*
bootstrap_files	*–bootstrap-file*	`[]`	*path l*
bootstrap_python_packages	*–bootstrap-python-package*	`[]`	*path l*
bootstrap_scripts	*–bootstrap-script*	`[]`	*path l*
check_emr_status_every	*–check-emr-status-every*	30	*string*
ec2_core_instance_bid_price	*–ec2-core-instance-bid-price*	`None`	*string*
ec2_core_instance_type	*–ec2-core-instance-type*	`'m1.small'`	*string*
ec2_instance_type	*–ec2-instance-type*	`'m1.small'`	*string*
ec2_key_pair	*–ec2-key-pair*	`None`	*string*
ec2_key_pair_file	*–ec2-key-pair-file*	`None`	*path*
ec2_master_instance_bid_price	*–ec2-master-instance-bid-price*	`None`	*string*
ec2_master_instance_type	*–ec2-master-instance-type*	`'m1.small'`	*string*
ec2_slave_instance_type	*–ec2-slave-instance-type*	value of *ec2_core_instance_type*	*string*
ec2_task_instance_bid_price	*–ec2-task-instance-bid-price*	`None`	*string*
ec2_task_instance_type	*–ec2-task-instance-type*	value of *ec2_core_instance_type*	*string*
emr_api_params	*–emr-api-param, –no-emr-api-param*	`{}`	*dict*
emr_endpoint	*–emr-endpoint*	infer from *aws_region*	*string*
emr_job_flow_id	*–emr-job-flow-id*	automatically create a job flow and use it	*string*
emr_job_flow_pool_name	*–emr-job-flow-pool-name*	`'default'`	*string*
enable_emr_debugging	*–enable-emr-debugging*	`False`	boolea
hadoop_streaming_jar_on_emr	*–hadoop-streaming-jar-on-emr*	AWS default	*string*
hadoop_version	*–hadoop-version*	inferred from environment/AWS	*string*
iam_job_flow_role	*–iam-job-flow-role*	`None`	*string*
max_hours_idle	*–max-hours-idle*	`None`	*string*
mins_to_end_of_hour	*–mins-to-end-of-hour*	5.0	*string*
num_ec2_core_instances	*–num-ec2-core-instances*	0	*string*
num_ec2_instances	*–num-ec2-instances*	1	*string*
num_ec2_task_instances	*–num-ec2-task-instances*	0	*string*
pool_emr_job_flows	*–pool-emr-job-flows*	`False`	*string*
pool_wait_minutes	*–pool-wait-minutes*	0	*string*
s3_endpoint	*–s3-endpoint*	infer from *aws_region*	*string*

Continued on next p

Table 1.1 – continued from previous page

Config	Command line	Default	Typ
s3_log_uri	*–s3-log-uri*	append `logs` to *s3_scratch_uri*	*string*
s3_scratch_uri	*–s3-scratch-uri*	`tmp/mrjob` in the first bucket belonging to you	*string*
s3_sync_wait_time	*–s3-sync-wait-time*	5.0	*string*
ssh_bin	*–ssh-bin*	`'ssh'`	*comma*
ssh_bind_ports	*–ssh-bind-ports*	`[40001, ..., 40840]`	specia
ssh_tunnel_is_open	*–ssh-tunnel-is-open*	`False`	boolea
ssh_tunnel_to_job_tracker	*–ssh-tunnel-to-job-tracker*	`False`	boolea
visible_to_all_users	*–visible-to-all-users*	`True`	boolea

1.9.5 Additional options for `HadoopJobRunner`

Config	Command line	Default	Type
check_input_paths	*–check-input-paths, –no-check-input-paths*	`True`	boolean
hadoop_bin	*–hadoop-bin*	*hadoop_home* plus `bin/hadoop`	*command*
hadoop_home	*–hadoop-home*	`HADOOP_HOME`	*path*
hdfs_scratch_dir	*–hdfs-scratch-dir*	`tmp/`	*path*

1.10 Job Environment Setup Cookbook

Many jobs have significant external dependencies, both libraries and other source code.

Combining shell syntax with Hadoop's DistributedCache notation, mrjob's *setup* option provides a powerful, dynamic alternative to pre-installing your Hadoop dependencies on every node.

All our `mrjob.conf` examples below are for the `hadoop` runner, but these work equally well with the `emr` runner. Also, if you are using EMR, take a look at the *EMR Bootstrapping Cookbook*.

1.10.1 Putting your source tree in `PYTHONPATH`

First you need to make a tarball of your source tree. Make sure that the root of your source tree is at the root of the tarball's file listing (e.g. the module `foo.bar` appears as `foo/bar.py` and not `your-src-dir/foo/bar.py`).

For reference, here is a command line that will put an entire source directory into a tarball:

```
tar -C your-src-code -f your-src-code.tar.gz -z -c .
```

Then, run your job with:

```
--setup 'export PYTHONPATH=$PYTHONPATH:your-src-dir.tar.gz#/'
```

If every job you run is going to want to use `your-src-code.tar.gz`, you can do this in your `mrjob.conf`:

```
runners:
  hadoop:
    setup:
    - export PYTHONPATH=$PYTHONPATH:your-src-code.tar.gz#/
```

1.10.2 Running a makefile inside your source dir

```
--setup 'cd your-src-dir.tar.gz#/' --setup 'make'
```

or, in mrjob.conf:

```
runners:
  hadoop:
    setup:
    - cd your-src-dir.tar.gz#
    - make
```

If Hadoop runs multiple tasks on the same node, your source dir will be shared between them. This is not a problem; mrjob automatically adds locking around setup commands to ensure that multiple copies of your setup script don't run simultaneously.

1.10.3 Making data files available to your job

Best practice for one or a few files is to use passthrough options; see `add_passthrough_option()`.

You can also use *upload_files* to upload file(s) into a task's working directory (or *upload_archives* for tarballs and other archives).

If you're a *setup* purist, you can also do something like this:

```
--setup 'true your-file#desired-name'
```

since **true** has no effect and ignores its arguments.

1.10.4 Using a virtualenv

What if you can't install the libraries you need on your Hadoop cluster?

You could do something like this in your `mrjob.conf`:

```
runners:
  hadoop:
    setup:
    - virtualenv venv
    - . venv/bin/activate
    - pip install mr3po simplejson
```

However, now the locking feature that protects **make** becomes a liability; each task on the same node has its own virtualenv, but one task has to finish setting up before the next can start.

The solution is to share the virtualenv between all tasks on the same machine, something like this:

```
runners:
  hadoop:
    setup:
    - VENV=/tmp/$mapreduce_job_id
    - if [ ! -e $VENV ]; then virtualenv $VENV; fi
    - . $VENV/bin/activate
    - pip install mr3po simplejson
```

With older versions of Hadoop (0.20 and earlier, and the 1.x series), you'd want to use `$mapred_job_id`.

1.10.5 Other ways to use pip to install Python packages

If you have a lot of dependencies, best practice is to make a pip requirements file and use the `-r` switch:

```
--setup 'pip install -r path/to/requirements.txt#'
```

Note that **pip** can also install from tarballs (which is useful for custom-built packages):

```
--setup 'pip install $MY_PYTHON_PKGS/*.tar.gz#'
```

1.11 Hadoop Cookbook

1.11.1 Increasing the task timeout

Warning: Some EMR AMIs appear to not support setting parameters like timeout with `jobconf` at run time. Instead, you must use *Bootstrap-time configuration*.

If your mappers or reducers take a long time to process a single step, you may want to increase the amount of time Hadoop lets them run before failing them as timeouts. You can do this with `jobconf` and the version-appropriate Hadoop environment variable. For example, this configuration will set the timeout to one hour:

```
runners:
  hadoop: # this will work for both hadoop and emr
    jobconf:
      # Hadoop 0.18
      mapred.task.timeout: 3600000
      # Hadoop 0.21+
      mapreduce.task.timeout: 3600000
```

mrjob will convert your `jobconf` options between Hadoop versions if necessary. In this example, either `jobconf` line could be removed and the timeout would still be changed when using either version of Hadoop.

1.11.2 Writing compressed output

To save space, you can have Hadoop automatically save your job's output as compressed files. This can be done using the same method as changing the task timeout, with `jobconf` and the appropriate environment variables:

```
runners:
  hadoop: # this will work for both hadoop and emr
    jobconf:
      # "true" must be a string argument, not a boolean! (#323)
      mapreduce.output.compress: "true"
      mapreduce.output.compression.codec: org.apache.hadoop.io.compress.BZip2Codec
```

1.12 Testing jobs

1.12.1 Inline vs local runner

The `inline` runner is the default runner for mrjob 0.4 and later. It runs your job in the same process as the runner so that you get faster feedback and simpler tracebacks.

The `local` runner runs your job in subprocesses in another directory and simulates several features of Hadoop, including:

- Multiple concurrent tasks
- `mapreduce.job.cache.archives`
- `mapreduce.job.cache.files`
- `mapreduce.job.cache.local.archives`
- `mapreduce.job.cache.local.files`
- `mapreduce.job.id`
- `mapreduce.job.local.dir`
- `mapreduce.map.input.file`
- `mapreduce.map.input.length`
- `mapreduce.map.input.start`
- `mapreduce.task.attempt.id`
- `mapreduce.task.id`
- `mapreduce.task.ismap`
- `mapreduce.task.output.dir`
- `mapreduce.task.partition`

If you specify *hadoop_version* <= 0.18, the simulated environment variables will change to use the names corresponding with the older Hadoop version.

See `LocalMRJobRunner` for reference about its behavior.

1.12.2 Anatomy of a test case

mrjob's test cases use the `unittest2` module, which is available for Python 2.3 and up. Most tests also require the `with` statement.

```
from __future__ import with_statement

try:
    import unittest2 as unittest
except ImportError:
    import unittest
```

You probably have your own job to test, but for this example we'll use a test of the `*_init()` methods from the mrjob test cases:

```
from mrjob.job import MRJob


class MRInitJob(MRJob):

    def __init__(self, *args, **kwargs):
        super(MRInitJob, self).__init__(*args, **kwargs)
        self.sum_amount = 0
        self.multiplier = 0
        self.combiner_multipler = 1
```

```
    def mapper_init(self):
        self.sum_amount += 10

    def mapper(self, key, value):
        yield(None, self.sum_amount)

    def reducer_init(self):
        self.multiplier += 10

    def reducer(self, key, values):
        yield(None, sum(values) * self.multiplier)

    def combiner_init(self):
        self.combiner_multiplier = 2

    def combiner(self, key, values):
        yield(None, sum(values) * self.combiner_multiplier)
```

Without using any mrjob features, we can write a simple test case to make sure our methods are behaving as expected:

```
class MRInitTestCase(unittest.TestCase):

    def test_mapper(self):
        j = MRInitJob()
        j.mapper_init()
        self.assertEqual(j.mapper(None, None).next(), (None, j.sum_amount))
```

To test the full job, you need to set up input, run the job, and check the collected output. The most straightforward way to provide input is to use the `sandbox()` method. Create a `StringIO` object, populate it with data, initialize your job to read from stdin, and enable the sandbox with your `StringIO` as stdin.

The simplest way to test the full job is with the `inline` runner. It runs the job in the same process as the test, so small jobs tend to run faster and stack traces are simpler. You'll probably also want to specify `--no-conf` so options from your local `mrjob.conf` don't pollute your testing environment.

This example reads from **stdin** (hence the − parameter):

```
def test_init_funcs(self):
    num_inputs = 2
    stdin = StringIO("x\n" * num_inputs)
    mr_job = MRInitJob(['-r', 'inline', '--no-conf', '-'])
    mr_job.sandbox(stdin=stdin)
```

To run the job without leaving temp files on your system, use the `make_runner()` context manager. `make_runner()` creates the runner specified in the command line arguments and ensures that job cleanup is performed regardless of the success or failure of the job.

Run the job with `run()`. The output lines are available as a generator through `stream_output()` and can be interpreted through the job's output protocol with `parse_output_line()`. You may choose to collect these lines in a list and check the contents of the list.

> **Warning:** Do not let your tests depend on the input lines being processed in a certain order. Input is divided nondeterministically by the `local`, `hadoop`, and `emr` runners.

```
results = []
with mr_job.make_runner() as runner:
    runner.run()
    for line in runner.stream_output():
        # Use the job's specified protocol to read the output
```

```
        key, value = mr_job.parse_output_line(line)
        results.append(value)

# these numbers should match if mapper_init, reducer_init, and
# combiner_init were called as expected
self.assertEqual(results[0], num_inputs * 10 * 10 * 2)
```

You should be able to switch out the `inline` runner for the `local` runner without changing any other code. The `local` runner will launch multiple subprocesses to run your job, which may expose assumptions about input order or race conditions.

1.13 Elastic MapReduce

1.13.1 Elastic MapReduce Quickstart

Configuring AWS credentials

Configuring your AWS credentials allows mrjob to run your jobs on Elastic MapReduce and use S3.

- Create an Amazon Web Services account
- Sign up for Elastic MapReduce
- Get your access and secret keys (click "Security Credentials" on your account page)

Now you can either set the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, or set **aws_access_key_id** and **aws_secret_access_key** in your `mrjob.conf` file like this:

```
runners:
  emr:
    aws_access_key_id: <your key ID>
    aws_secret_access_key: <your secret>
```

Configuring SSH credentials

Configuring your SSH credentials lets mrjob open an SSH tunnel to your jobs' master nodes to view live progress, see the job tracker in your browser, and fetch error logs quickly.

- Go to https://console.aws.amazon.com/ec2/home
- Make sure the **Region** dropdown (upper left) matches the region you want to run jobs in (usually "US East").
- Click on **Key Pairs** (lower left)
- Click on **Create Key Pair** (center).
- Name your key pair `EMR` (any name will work but that's what we're using in this example)
- Save `EMR.pem` wherever you like (`~/.ssh` is a good place)
- Run `chmod og-rwx /path/to/EMR.pem` so that `ssh` will be happy
- Add the following entries to your `mrjob.conf`:

```
runners:
  emr:
    ec2_key_pair: EMR
    ec2_key_pair_file: /path/to/EMR.pem # ~/ and $ENV_VARS allowed here
    ssh_tunnel_to_job_tracker: true
```

Running an EMR Job

Running a job on EMR is just like running it locally or on your own Hadoop cluster, with the following changes:

- The job and related files are uploaded to S3 before being run
- The job is run on EMR (of course)
- Output is written to S3 before mrjob streams it to stdout locally
- The Hadoop version is specified by the EMR AMI version

This the output of this command should be identical to the output shown in *Fundamentals*, but it should take much longer:

> python word_count.py -r emr README.txt "chars" 3654 "lines" 123 "words" 417

Sending Output to a Specific Place

If you'd rather have your output go to somewhere deterministic on S3, which you probably do, use `--output-dir`:

```
> python word_count.py -r emr README.rst \
>   --output-dir=s3://my-bucket/wc_out/
```

It's also likely that since you know where your output is on S3, you don't want output streamed back to your local machine. For that, use `--no-output`:

```
> python word_count.py -r emr README.rst \
>   --output-dir=s3://my-bucket/wc_out/ \
>   --no-output
```

There are many other ins and outs of effectively using mrjob with EMR. See *Advanced EMR usage* for some of the ins, but the outs are left as an exercise for the reader. This is a strictly no-outs body of documentation!

Choosing Type and Number of EC2 Instances

When you create a job flow on EMR, you'll have the option of specifying a number and type of EC2 instances, which are basically virtual machines. Each instance type has different memory, CPU, I/O and network characteristics, and costs a different amount of money. See Instance Types and Pricing for details.

Instances perform one of three roles:

- **Master**: There is always one master instance. It handles scheduling of tasks (i.e. mappers and reducers), but does not run them itself.
- **Core**: You may have one or more core instances. These run tasks and host HDFS.
- **Task**: You may have zero or more of these. These run tasks, but do *not* host HDFS. This is mostly useful because your job flow can lose task instances without killing your job (see *Spot Instances*).

There's a special case where your job flow *only* has a single master instance, in which case the master instance schedules tasks, runs them, and hosts HDFS.

By default, `mrjob` runs a single `m1.small`, which is a cheap but not very powerful instance type. This can be quite adequate for testing your code on a small subset of your data, but otherwise give little advantage over running a job locally. To get more performance out of your job, you can either add more instances, use more powerful instances, or both.

Here are some things to consider when tuning your instance settings:

- Amazon bills you for the full hour even if your job flow only lasts for a few minutes (this is an artifact of the EC2 billing structure), so for many jobs that you run repeatedly, it is a good strategy to pick instance settings that make your job consistently run in a little less than an hour.
- Your job will take much longer and may fail if any task (usually a reducer) runs out of memory and starts using swap. (You can verify this by using **vmstat** with `mrboss`.) Restructuring your job is often the best solution, but if you can't, consider using a high-memory instance type.
- Larger instance types are usually a better deal if you have the workload to justify them. For example, a `c1.xlarge` costs about 10 times as much as an `m1.small`, but it has about 20 times as much processing power (and more memory).

The basic way to control type and number of instances is with the *ec2_instance_type* and *num_ec2_instances* options, on the command line like this:

```
--ec2-instance-type c1.medium --num-ec2-instances 5
```

or in `mrjob.conf`, like this:

```
runners:
  emr:
    ec2_instance_type: c1.medium
    num_ec2_instances: 5
```

In most cases, your master instance type doesn't need to be larger than''m1.small'' to schedule tasks, so *ec2_instance_type* only applies to instances that actually run tasks. (In this example, there are 1 `m1.small` master instance, and 4 `c1.medium` core instances.) You *will* need a larger master instance if you have a very large number of input files; in this case, use the *ec2_master_instance_type* option.

If you want to run task instances, you instead must specify the number of core and task instances directly with the *num_ec2_core_instances* and *num_ec2_task_instances* options. There are also *ec2_core_instance_type* and *ec2_task_instance_type* options if you want to set these directly.

1.13.2 EMR runner options

All options from *Options available to all runners* and *Hadoop-related options* are available to the emr runner.

Amazon credentials

See *Configuring AWS credentials* and *Configuring SSH credentials* for specific instructions about setting these options.

aws_access_key_id (`--aws-access-key-id`) [*string*] **Default:** `None`

"username" for Amazon web services.

aws_secret_access_key (`--aws-secret-access-key`) [*string*] **Default:** `None`

your "password" on AWS

ec2_key_pair (`--ec2-key-pair`) [*string*] **Default:** `None`

name of the SSH key you set up for EMR.

ec2_key_pair_file (`--ec2-key-pair-file`) [*path*] **Default:** `None`

path to file containing the SSH key for EMR

Job flow creation and configuration

additional_emr_info (`--additional-emr-info`) [special] **Default:** `None`

Special parameters to select additional features, mostly to support beta EMR features. Pass a JSON string on the command line or use data structures in the config file (which is itself basically JSON).

ami_version (`--ami-version`) [*string*] **Default:** `'latest'`

EMR AMI version to use. This controls which Hadoop version(s) are available and which version of Python is installed, among other things; see the AWS docs on specifying the AMI version. for details.

aws_availability_zone (`--aws-availability-zone`) [*string*] **Default:** AWS default

Availability zone to run the job in

aws_region (`--aws-region`) [*string*] **Default:** infer from scrach bucket region

region to connect to S3 and EMR on (e.g. `us-west-1`). If you want to use separate regions for S3 and EMR, set *emr_endpoint* and *s3_endpoint*.

emr_endpoint (`--emr-endpoint`) [*string*] **Default:** infer from *aws_region*

optional host to connect to when communicating with S3 (e.g. `us-west-1.elasticmapreduce.amazonaws.com`).

hadoop_streaming_jar_on_emr (`--hadoop-streaming-jar-on-emr`) [*string*] **Default:** AWS default

Like *hadoop_streaming_jar*, except that it points to a path on the EMR instance, rather than to a local file or one on S3. Rarely necessary to set this by hand.

iam_job_flow_role (`--iam-job-flow-role`) [*string*] **Default:** `None`

IAM job flow role to use on the EMR cluster. See http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-iam-roles.html for more details on using IAM roles with EMR. Needs AMI version 2.3.0 or later to work.

New in version 0.4.3.

max_hours_idle (`--max-hours-idle`) [*string*] **Default:** `None`

If we create a persistent job flow, have it automatically terminate itself after it's been idle this many hours AND we're within *mins_to_end_of_hour* of an EC2 billing hour.

New in version 0.4.1.

mins_to_end_of_hour (`--mins-to-end-of-hour`) [*string*] **Default:** 5.0

If *max_hours_idle* is set, controls how close to the end of an EC2 billing hour the job flow can automatically terminate itself.

New in version 0.4.1.

visible_to_all_users (`--visible-to-all-users`) [boolean] **Default:** `True`

If `True`, EMR job flows will be visible to all IAM users. If `False`, the job flow will only be visible to the IAM user that created it. This parameter can be overridden by *emr_api_params* with key `VisibleToAllUsers`.

New in version 0.4.1.

emr_api_params (`--emr-api-param`, `--no-emr-api-param`) [*dict*] **Default:** `{}`

Additional parameters to pass directly to the EMR API. This is a way to pass new API parameters without updating the library. If value for parameter is set to `None`, this parameter will be deleted from API call.

Bootstrapping

These options apply at *bootstrap time*, before the Hadoop cluster has started. Bootstrap time is a good time to install Debian packages or compile and install another Python binary.

bootstrap (`--bootstrap`) [*string list*] **Default:** `[]`

A list of lines of shell script to run once on each node in your job flow, at bootstrap time.

This option is complex and powerful; the best way to get started is to read the *EMR Bootstrapping Cookbook*.

Passing expressions like `path#name` will cause *path* to be automatically uploaded to the task's working directory with the filename *name*, marked as executable, and interpolated into the script by their absolute path on the machine running the script. *path* may also be a URI, and ~ and environment variables within *path* will be resolved based on the local environment. *name* is optional. For details of parsing, see `parse_setup_cmd()`.

Unlike with *setup*, archives are not supported (unpack them yourself).

Remember to put `sudo` before commands requiring root privileges!

bootstrap_actions (`--bootstrap-actions`) [*string list*] **Default:** `[]`

A list of raw bootstrap actions (essentially scripts) to run prior to any of the other bootstrap steps. Any arguments should be separated from the command by spaces (we use `shlex.split()`). If the action is on the local filesystem, we'll automatically upload it to S3.

This has little advantage over *bootstrap*; it is included in order to give direct access to the EMR API.

bootstrap_cmds (`--bootstrap-cmd`) [*string list*] **Default:** `[]`

Deprecated since version 0.4.2.

A list of commands to run at bootstrap time. Basically *bootstrap* without automatic file uploading/interpolation. Can also take commands as lists of arguments.

bootstrap_files (`--bootstrap-file`) [*path list*] **Default:** `[]`

Deprecated since version 0.4.2.

Files to download to the bootstrap working directory before running bootstrap commands. Use the *bootstrap* option's file auto-upload/interpolation feature instead.

bootstrap_python_packages (`--bootstrap-python-package`) [*path list*] **Default:** `[]`

Deprecated since version 0.4.2.

Paths of python modules tarballs to install on EMR. Pass `pip install path/to/tarballs/*.tar.gz#` to *bootstrap* instead.

bootstrap_scripts (`--bootstrap-script`) [*path list*] **Default:** `[]`

Deprecated since version 0.4.2.

Scripts to upload and then run at bootstrap time. Pass `path/to/script# args` to *bootstrap* instead.

Monitoring the job flow

check_emr_status_every (`--check-emr-status-every`) [*string*] **Default:** 30

How often to check on the status of EMR jobs in seconds. If you set this too low, AWS will throttle you.

enable_emr_debugging (`--enable-emr-debugging`) [boolean] **Default:** `False`

store Hadoop logs in SimpleDB

Number and type of instances

ec2_instance_type (--ec2-instance-type) [*string*] **Default:** `'m1.small'`

What sort of EC2 instance(s) to use on the nodes that actually run tasks (see http://aws.amazon.com/ec2/instance-types/). When you run multiple instances (see *num_ec2_instances*), the master node is just coordinating the other nodes, so usually the default instance type (`m1.small`) is fine, and using larger instances is wasteful.

ec2_core_instance_type (--ec2-core-instance-type) [*string*] **Default:** `'m1.small'`

like *ec2_instance_type*, but only for the core (also know as "slave") Hadoop nodes; these nodes run tasks and host HDFS. Usually you just want to use *ec2_instance_type*.

ec2_core_instance_bid_price (--ec2-core-instance-bid-price) [*string*] **Default:** `None`

When specified and not "0", this creates the master Hadoop node as a spot instance at this bid price. You usually only want to set bid price for task instances.

ec2_master_instance_type (--ec2-master-instance-type) [*string*] **Default:** `'m1.small'`

like *ec2_instance_type*, but only for the master Hadoop node. This node hosts the task tracker and HDFS, and runs tasks if there are no other nodes. Usually you just want to use *ec2_instance_type*.

ec2_master_instance_bid_price (--ec2-master-instance-bid-price) [*string*] **Default:** `None`

When specified and not "0", this creates the master Hadoop node as a spot instance at this bid price. You usually only want to set bid price for task instances unless the master instance is your only instance.

ec2_slave_instance_type (--ec2-slave-instance-type) [*string*] **Default:** value of *ec2_core_instance_type*

An alias for *ec2_core_instance_type*, for consistency with the EMR API.

ec2_task_instance_type (--ec2-task-instance-type) [*string*] **Default:** value of *ec2_core_instance_type*

like *ec2_instance_type*, but only for the task Hadoop nodes; these nodes run tasks but do not host HDFS. Usually you just want to use *ec2_instance_type*.

ec2_task_instance_bid_price (--ec2-task-instance-bid-price) [*string*] **Default:** `None`

When specified and not "0", this creates the master Hadoop node as a spot instance at this bid price. (You usually only want to set bid price for task instances.)

num_ec2_core_instances (--num-ec2-core-instances) [*string*] **Default:** 0

Number of core (or "slave") instances to start up. These run your job and host HDFS. Incompatible with *num_ec2_instances*. This is in addition to the single master instance.

num_ec2_instances (--num-ec2-instances) [*string*] **Default:** 1

Total number of instances to start up; basically the number of core instance you want, plus 1 (there is always one master instance). Incompatible with *num_ec2_core_instances* and *num_ec2_task_instances*.

num_ec2_task_instances (--num-ec2-task-instances) [*string*] **Default:** 0

Number of task instances to start up. These run your job but do not host HDFS. Incompatible with *num_ec2_instances*. If you use this, you must set *num_ec2_core_instances*; EMR does not allow you to run task instances without core instances (because there's nowhere to host HDFS).

Choosing/creating a job flow to join

emr_job_flow_id (--emr-job-flow-id) [*string*] **Default:** automatically create a job flow and use it

The ID of a persistent EMR job flow to run jobs in. It's fine for other jobs to be using the job flow; we give our job's steps a unique ID.

emr_job_flow_pool_name (`--emr-job-flow-pool-name`) [*string*] **Default:** `'default'`

Specify a pool name to join. Does not imply `pool_emr_job_flows`.

pool_emr_job_flows (`--pool-emr-job-flows`) [*string*] **Default:** `False`

Try to run the job on a `WAITING` pooled job flow with the same bootstrap configuration. Prefer the one with the most compute units. Use S3 to "lock" the job flow and ensure that the job is not scheduled behind another job. If no suitable job flow is *WAITING*, create a new pooled job flow.

> **Warning:** Do not run this without either setting *max_hours_idle* or putting `mrjob.tools.emr.terminate.idle_job_flows` in your crontab; job flows left idle can quickly become expensive!

pool_wait_minutes (`--pool-wait-minutes`) [*string*] **Default:** 0

If pooling is enabled and no job flow is available, retry finding a job flow every 30 seconds until this many minutes have passed, then start a new job flow instead of joining one.

S3 paths and options

s3_endpoint (`--s3-endpoint`) [*string*] **Default:** infer from *aws_region*

Host to connect to when communicating with S3 (e.g. `s3-us-west-1.amazonaws.com`).

s3_log_uri (`--s3-log-uri`) [*string*] **Default:** append `logs` to *s3_scratch_uri*

Where on S3 to put logs, for example `s3://yourbucket/logs/`. Logs for your job flow will go into a subdirectory, e.g. `s3://yourbucket/logs/j-JOBFLOWID/`. in this example s3://yourbucket/logs/j-YOURJOBID/).

s3_scratch_uri (`--s3-scratch-uri`) [*string*] **Default:** `tmp/mrjob` in the first bucket belonging to you

S3 directory (URI ending in `/`) to use as scratch space, e.g. `s3://yourbucket/tmp/`.

s3_sync_wait_time (`--s3-sync-wait-time`) [*string*] **Default:** 5.0

How long to wait for S3 to reach eventual consistency. This is typically less than a second (zero in U.S. West), but the default is 5.0 to be safe.

SSH access and tunneling

ssh_bin (`--ssh-bin`) [*command*] **Default:** `'ssh'`

Path to the ssh binary; may include switches (e.g. `'ssh -v'` or `['ssh', '-v']`). Defaults to **ssh**

ssh_bind_ports (`--ssh-bind-ports`) [special] **Default:** `[40001, ..., 40840]`

A list of ports that are safe to listen on. The command line syntax looks like `2000[:2001][,2003,2005:2008,etc]`, where commas separate ranges and colons separate range endpoints.

ssh_tunnel_to_job_tracker (`--ssh-tunnel-to-job-tracker`) [boolean] **Default:** `False`

If True, create an ssh tunnel to the job tracker and listen on a randomly chosen port. This requires you to set *ec2_key_pair* and *ec2_key_pair_file*. See *Configuring SSH credentials* for detailed instructions.

ssh_tunnel_is_open (`--ssh-tunnel-is-open`) [boolean] **Default:** `False`

if True, any host can connect to the job tracker through the SSH tunnel you open. Mostly useful if your browser is running on a different machine from your job runner.

1.13.3 EMR Bootstrapping Cookbook

Bootstrapping allows you to configure EMR machines to your needs.

AMI 2.x and AMI 3.x version differences

AMI versions 2.x are based on Debian 6.0.2 (Squeeze). The package management system is `apt-get`. Any package distributed with Debian Squeeze should be available.

AMI versions 3.x are based on Amazon Linux Release 2012.09. This major bump changed the package management system to `yum`. You can view the list of RPM packages Amazon distributed with the 2012.09.1 release here.

You can follow the AMI changelog here.

Installing Python packages with pip

First you need to install **pip**.

For AMI 2.x versions use `apt-get`:

```
--bootstrap 'sudo apt-get install -y python-pip'
```

For AMI 3.x versions use `yum`:

```
--bootstrap 'sudo yum install -y python-pip'
```

Then install the packages you want:

```
--bootstrap 'sudo pip install --upgrade mr3po simplejson'
```

To support both AMI 2.x and AMI 3.x:

```
--bootstrap 'sudo apt-get install -y python-pip || sudo yum install -y python-pip'
```

Or, equivalently, in `mrjob.conf`:

```
runners:
  emr:
    bootstrap:
    - sudo apt-get install -y python-pip || sudo yum install -y python-pip
    - sudo pip install boto mr3po
```

Upgrading simplejson

mrjob relies on simplejson for rapid encoding and decoding of data.

To use the latest (fastest) version, do:

```
--bootstrap 'sudo pip install --upgrade simplejson'
```

Other ways to use pip to install Python packages

If you have a lot of dependencies, best practice is to make a pip requirements file and use the `-r` switch:

```
--bootstrap 'sudo pip install -r path/to/requirements.txt#'
```

Note that **pip** can also install from tarballs (which is useful for custom-built packages):

```
--bootstrap 'sudo pip install $MY_PYTHON_PKGS/*.tar.gz#'
```

Installing Debian packages on AMI 2.x:

As we did with **pip**, you can use `apt-get` to install any package from the Debian archive. For example, to install Python 3:

```
--bootstrap 'sudo apt-get install -y python3'
```

If you have particular `.deb` files you want to install, do:

```
--bootstrap 'sudo dpkg -i path/to/packages/*.deb#'
```

Installing RPM Packages on AMI 3.x:

Conversely, while running on an AMI 3.x you can install the Python 3 RPM archive by using `yum`:

```
--bootstrap 'sudo yum install -y python3'
```

Likewise, if you have a particular `.rpm` files you want to install, do:

```
--bootstrap 'sudo yum install -y path/to/packages/*.rpm#'
```

Upgrading Python from source

To upgrade Python on EMR, you will probably have to build it from source (Debian packages tend to lag the current versions of software, and EMR AMIs tend to lag the current version of Debian).

First, download the latest version of the Python source here.

Then add this to your `mrjob.conf`:

```
runners:
  emr:
    bootstrap:
    - tar xfz path/to/Python-x.y.z.tgz#
    - cd Python-x.y.z
    - ./configure && make && sudo make install
```

bootstrap_mrjob runs *last*, so mrjob will get bootstrapped into your newly upgraded version of Python. If you use other bootstrap commands to install/upgrade Python libraries, you should also run them *after* upgrading Python.

When to use bootsrap, and when to use setup

You can use *bootstrap* and *setup* together.

Generally, you want to use *bootstrap* for things that are part of your general production environment, and *setup* for things that are specific to your particular job. This makes things work as expected if you are *Pooling Job Flows*.

1.13.4 Troubleshooting

Many things can go wrong in an EMR job, and the system's distributed nature can make it difficult to find the source of a problem. `mrjob` attempts to simplify the debugging process by automatically scanning logs for probable causes of failure. Specifically, it looks at logs relevant to your job for these errors:

- Python tracebacks and Java stack traces in `$S3_LOG_URI/task-attempts/*`
- Hadoop Streaming errors in `$S3_LOG_URI/steps/*`
- Timeout errors in `$S3_LOG_URI/jobs/*`

As mentioned above, in addition to looking at S3, `mrjob` can be configured to also use SSH to fetch error logs directly from the master and slave nodes. This can speed up debugging significantly, because logs are only available on S3 five minutes after the job completes, or immediately after the job flow terminates.

Using persistent job flows

When troubleshooting a job, it can be convenient to use a persistent job flow to avoid having to wait for bootstrapping every run. **If you decide to use persistent job flows, add `mrjob.tools.emr.terminate_idle_job_flows` to your crontab or you will be billed for unused CPU time if you forget to explicitly terminate job flows.**

First, use the `mrjob.tools.emr.create_job_flow` tool to create a persistent job flow:

```
> python -m mrjob.tools.emr.create_job_flow
using configs in /etc/mrjob.conf
Creating persistent job flow to run several jobs in...
creating tmp directory /scratch/username/no_script.username.20110811.185141.422311
writing master bootstrap script to /scratch/username/no_script.username.20110811.185141.422311/b.py
Copying non-input files into s3://scratch-bucket/tmp/no_script.username.20110811.185141.422311/files/
Waiting 5.0s for S3 eventual consistency
Creating Elastic MapReduce job flow
Job flow created with ID: j-1NXMMBNEQHAFT
j-1NXMMBNEQHAFT
```

Now you can use the job flow ID to start the troublesome job:

```
> python mrjob/buggy_job.py -r emr --emr-job-flow-id=j-1NXMMBNEQHAFT input/* > out
using configs in /etc/mrjob.conf
Uploading input to s3://scratch-bucket/tmp/buggy_job.username.20110811.185410.536519/input/
creating tmp directory /scratch/username/buggy_job.username.20110811.185410.536519
writing wrapper script to /scratch/username/buggy_job.username.20110811.185410.536519/wrapper.py
Copying non-input files into s3://scratch-bucket/tmp/buggy_job.username.20110811.185410.536519/files/
Adding our job to job flow j-1NXMMBNEQHAFT
Job launched 30.1s ago, status BOOTSTRAPPING: Running bootstrap actions
Job launched 60.1s ago, status BOOTSTRAPPING: Running bootstrap actions
Job launched 90.2s ago, status BOOTSTRAPPING: Running bootstrap actions
Job launched 120.3s ago, status BOOTSTRAPPING: Running bootstrap actions
Job launched 150.3s ago, status BOOTSTRAPPING: Running bootstrap actions
Job launched 180.4s ago, status RUNNING: Running step (buggy_job.username.20110811.185410.536519: Ste
Opening ssh tunnel to Hadoop job tracker
Connect to job tracker at: http://dev5sj.sjc.yelpcorp.com:40753/jobtracker.jsp
Job launched 211.5s ago, status RUNNING: Running step (buggy_job.username.20110811.185410.536519: Ste
 map 100% reduce   0%
Job launched 241.8s ago, status RUNNING: Running step (buggy_job.username.20110811.185410.536519: Ste
 map   0% reduce   0%
Job launched 271.9s ago, status RUNNING: Running step (buggy_job.username.20110811.185410.536519: Ste
 map 100% reduce 100%
Job failed with status WAITING: Waiting after step failed
```

```
Logs are in s3://scratch-bucket/tmp/logs/j-1NXMMBNEQHAFT/
Scanning SSH logs for probable cause of failure
Probable cause of failure (from ssh://ec2-50-18-136-229.us-west-1.compute.amazonaws.com/mnt/var/log/h
Traceback (most recent call last):
  File "buggy_job.py", line 36, in <module>
    MRWordFreqCount.run()
  File "/usr/lib/python2.5/site-packages/mrjob/job.py", line 448, in run
    mr_job.execute()
  File "/usr/lib/python2.5/site-packages/mrjob/job.py", line 455, in execute
    self.run_mapper(self.options.step_num)
  File "/usr/lib/python2.5/site-packages/mrjob/job.py", line 548, in run_mapper
    for out_key, out_value in mapper(key, value) or ():
  File "buggy_job.py", line 24, in mapper
    raise IndexError
IndexError
Traceback (most recent call last):
  File "wrapper.py", line 16, in <module>
    check_call(sys.argv[1:])
  File "/usr/lib/python2.5/subprocess.py", line 462, in check_call
    raise CalledProcessError(retcode, cmd)
subprocess.CalledProcessError: Command '['python', 'buggy_job.py', '--step-num=0', '--mapper']' retur
(while reading from s3://scratch-bucket/tmp/buggy_job.username.20110811.185410.536519/input/00000-REA
Killing our SSH tunnel (pid 988)
...
```

The same job flow ID can be used to start new jobs without waiting for a new job flow to bootstrap.

Note that SSH must be set up for logs to be scanned from persistent jobs.

Finding failures after the fact

If you are trying to look at a failure after the original process has exited, you can use the `mrjob.tools.emr.fetch_logs` tool to scan the logs:

```
> python -m mrjob.tools.emr.fetch_logs --find-failure j-1NXMMBNEQHAFT
using configs in /etc/mrjob.conf
Scanning SSH logs for probable cause of failure
Probable cause of failure (from ssh://ec2-50-18-136-229.us-west-1.compute.amazonaws.com/mnt/var/log/h
Traceback (most recent call last):
  File "buggy_job.py", line 36, in <module>
    MRWordFreqCount.run()
  File "/usr/lib/python2.5/site-packages/mrjob/job.py", line 448, in run
    mr_job.execute()
  File "/usr/lib/python2.5/site-packages/mrjob/job.py", line 455, in execute
    self.run_mapper(self.options.step_num)
  File "/usr/lib/python2.5/site-packages/mrjob/job.py", line 548, in run_mapper
    for out_key, out_value in mapper(key, value) or ():
  File "buggy_job.py", line 24, in mapper
    raise IndexError
IndexError
Traceback (most recent call last):
  File "wrapper.py", line 16, in <module>
    check_call(sys.argv[1:])
  File "/usr/lib/python2.5/subprocess.py", line 462, in check_call
    raise CalledProcessError(retcode, cmd)
subprocess.CalledProcessError: Command '['python', 'buggy_job.py', '--step-num=0', '--mapper']' retur
(while reading from s3://scratch-bucket/tmp/buggy_job.username.20110811.185410.536519/input/00000-REA
Removing all files in s3://scratch-bucket/tmp/no_script.username.20110811.190217.810442/
```

Determining cause of failure when mrjob can't

In some cases, `mrjob` will be unable to find the reason your job failed, or it will report an error that was merely a symptom of a larger problem. You can look at the logs yourself by using Amazon's elastic-mapreduce tool to SSH to the master node:

```
> elastic-mapreduce --ssh j-1NXMMBNEQHAFT
ssh -i /nail/etc/EMR.pem.dev hadoop@ec2-50-18-136-229.us-west-1.compute.amazonaws.com
...
hadoop@ip-10-172-51-151:~$ grep --recursive 'Traceback' /mnt/var/log/hadoop
/mnt/var/log/hadoop/userlogs/attempt_201108111855_0001_m_000000_0/stderr:Traceback (most recent call
...
hadoop@ip-10-172-51-151:~$ cat /mnt/var/log/hadoop/userlogs/attempt_201108111855_0001_m_000000_0/stde
Exception exceptions.RuntimeError: 'generator ignored GeneratorExit' in <generator object at 0x94d57c
Traceback (most recent call last):
  File "mr_word_freq_count.py", line 36, in <module>
    MRWordFreqCount.run()
  File "/usr/lib/python2.5/site-packages/mrjob/job.py", line 448, in run
    mr_job.execute()
  File "/usr/lib/python2.5/site-packages/mrjob/job.py", line 455, in execute
    self.run_mapper(self.options.step_num)
  File "/usr/lib/python2.5/site-packages/mrjob/job.py", line 548, in run_mapper
    for out_key, out_value in mapper(key, value) or ():
  File "mr_word_freq_count.py", line 24, in mapper
    raise IndexError
IndexError
Traceback (most recent call last):
  File "wrapper.py", line 16, in <module>
    check_call(sys.argv[1:])
  File "/usr/lib/python2.5/subprocess.py", line 462, in check_call
    raise CalledProcessError(retcode, cmd)
subprocess.CalledProcessError: Command '['python', 'mr_word_freq_count.py', '--step-num=0', '--mapper
java.lang.RuntimeException: PipeMapRed.waitOutputThreads(): subprocess failed with code 1
    at org.apache.hadoop.streaming.PipeMapRed.waitOutputThreads(PipeMapRed.java:372)
    at org.apache.hadoop.streaming.PipeMapRed.mapRedFinished(PipeMapRed.java:582)
    at org.apache.hadoop.streaming.PipeMapper.close(PipeMapper.java:135)
    at org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:57)
    at org.apache.hadoop.streaming.PipeMapRunner.run(PipeMapRunner.java:36)
    at org.apache.hadoop.mapred.MapTask.runOldMapper(MapTask.java:363)
    at org.apache.hadoop.mapred.MapTask.run(MapTask.java:312)
    at org.apache.hadoop.mapred.Child.main(Child.java:170)
```

1.13.5 Advanced EMR usage

Reusing Job Flows

It can take several minutes to create a job flow. To decrease wait time when running multiple jobs, you may find it convenient to reuse a single job.

`mrjob` includes a utility to create persistent job flows without running a job. For example, this command will create a job flow with 12 EC2 instances (1 master and 11 slaves), taking all other options from `mrjob.conf`:

```
> python mrjob/tools/emr/create_job_flow.py --num-ec2-instances=12
...
Job flow created with ID: j-JOBFLOWID
```

You can then add jobs to the job flow with the *--emr-job-flow-id* switch or the *emr_job_flow_id* variable in *mrjob.conf* (see `EMRJobRunner.__init__()`):

```
> python mr_my_job.py -r emr --emr-job-flow-id=j-JOBFLOWID input_file.txt > out
...
Adding our job to job flow j-JOBFLOWID
...
```

Debugging will be difficult unless you complete SSH setup (see *Configuring SSH credentials*) since the logs will not be copied from the master node to S3 before either five minutes pass or the job flow terminates.

Pooling Job Flows

Manually creating job flows to reuse and specifying the job flow ID for every run can be tedious. In addition, it is not convenient to coordinate job flow use among multiple users.

To mitigate these problems, `mrjob` provides **job flow pools.** Rather than having to remember to start a job flow and copying its ID, simply pass *--pool-emr-job-flows* on the command line. The first time you do this, a new job flow will be created that does not terminate when the job completes. When you use *--pool-emr-job-flows* the next time, it will identify the job flow and add the job to it rather than creating a new one.

> **Warning:** If you use job flow pools, keep `terminate_idle_job_flows` in your crontab! Otherwise you may forget to terminate your job flows and waste a lot of money.

Alternatively, you may use the *max_hours_idle* option to create self-terminating job flows; the disadvantage is that pooled jobs may occasionally join job flows with out knowing they are about to self-terminate (this is better for development than production).

Pooling is designed so that jobs run against the same `mrjob.conf` can share the same job flows. This means that the version of `mrjob`, boostrap configuration, Hadoop version and AMI version all need to be exactly the same.

Pooled jobs will also only use job flows with the same **pool name**, so you can use the *--pool-name* option to partition your job flows into separate pools.

Pooling is flexible about instance type and number of instances; it will attempt to select the most powerful job flow available as long as the job flow's instances provide at least as much memory and at least as much CPU as your job requests. If there is a tie, it picks job flows that are closest to the end of a full hour, to minimize wasted instance hours.

Amazon limits job flows to 256 steps total; pooling respects this and won't try to use pooled job flows that are "full." `mrjob` also uses an S3-based "locking" mechanism to prevent two jobs from simultaneously joining the same job flow. This is somewhat ugly but works in practice, and avoids `mrjob` depending on Amazon services other than EMR and S3.

> **Warning:** If S3 eventual consistency takes longer than *s3_sync_wait_time*, then you may encounter race conditions when using pooling, e.g. two jobs claiming the same job flow at the same time, or the idle job flow killer shutting down your job before it has started to run. Regions with read-after-write consistency (i.e. every region except US Standard) should not experience these issues.

You can allow jobs to wait for an available job flow instead of immediately starting a new one by specifying a value for *–pool-wait-minutes*. mrjob will try to find a job flow every 30 seconds for **pool_wait_minutes**. If none is found during that time, mrjob will start a new one.

Spot Instances

Amazon also has a spot market for EC2 instances. You can potentially save money by using the spot market. The catch is that if someone bids more for instances that you're using, they can be taken away from your job flow. If this

happens, you aren't charged, but your job may fail.

You can specify spot market bid prices using the *ec2_core_instance_bid_price*, *ec2_master_instance_bid_price*, and *ec2_task_instance_bid_price* options to specify a price in US dollars. For example, on the command line:

```
--ec2-task-instance-bid-price 0.42
```

or in `mrjob.conf`:

```
runners:
  emr:
    ec2_task_instance_bid_price: '0.42'
```

(Note the quotes; bid prices are strings, not floats!)

Amazon has a pretty thorough explanation of why and when you'd want to use spot instances here. The brief summary is that either you don't care if your job fails, in which case you want to purchase all your instances on the spot market, or you'd need your job to finish but you'd like to save time and money if you can, in which case you want to run task instances on the spot market and purchase master and core instances the regular way.

Job flow pooling interacts with bid prices more or less how you'd expect; a job will join a pool with spot instances only if it requested spot instances at the same price or lower.

Custom Python packages

There are a couple of ways to install Python packages that are not in the standard library. If there is a Debian package and you are running on AMI 2.x, you can add a call to `apt-get` as a `bootstrap_cmd`:

```
runners:
  emr:
    bootstrap_cmds:
    - sudo apt-get install -y python-simplejson
```

Otherwise, if you are running on AMI 3.x and have an RPM package you would like to install you can use `yum` as a `bootstrap_cmd`:

```
runners:
  emr:
    bootstrap_cmds:
    - sudo yum install -y python-simplejson
```

If there is no Debian or RPM package or you prefer to use your own tarballs for some other reason, you can specify tarballs in `bootstrap_python_packages`, which supports glob syntax:

```
runners:
  emr:
    bootstrap_python_packages:
    - $MY_SOURCE_TREE/emr_packages/*.tar.gz
```

Bootstrap-time configuration

Some Hadoop options, such as the maximum number of running map tasks per node, must be set at bootstrap time and will not work with *–jobconf*. You must use Amazon's *configure-hadoop* script for this. For example, this limits the number of mappers and reducers to one per node:

```
--bootstrap-action="s3://elasticmapreduce/bootstrap-actions/configure-hadoop \
-m mapred.tasktracker.map.tasks.maximum=1 \
-m mapred.tasktracker.reduce.tasks.maximum=1"
```

Setting up Ganglia

Ganglia is a scalable distributed monitoring system for high-performance computing systems. You can enable it for your EMR cluster with Amazon's install-ganglia bootstrap action:

```
--bootstrap-action="s3://elasticmapreduce/bootstrap-actions/install-ganglia
```

Enabling Python core dumps

Particularly bad errors may leave no traceback in the logs. To enable core dumps on your EMR instances, put this script in `core_dump_bootstrap.sh`:

```
#!/bin/sh

chk_root () {
    if [ ! $( id -u ) -eq 0 ]; then
        exec sudo sh ${0}
        exit ${?}
    fi
}

chk_root

mkdir /tmp/cores
chmod -R 1777 /tmp/cores
echo "\n* soft core unlimited" >> /etc/security/limits.conf
echo "ulimit -c unlimited" >> /etc/profile
echo "/tmp/cores/core.%e.%p.%h.%t" > /proc/sys/kernel/core_pattern
```

Use the script as a bootstrap action in your job:

```
--bootstrap-action=core_dump_setup.sh
```

You'll probably want to use a version of Python with debugging symbols, so install it and use it as `python_bin`:

```
--bootstrap-cmd="sudo apt-get install -y python2.6-dbg" \
--python-bin=python2.6-dbg
```

Run your job in a persistent job flow. When it fails, you can SSH to your nodes to inspect the core dump files:

```
you@local: emr --ssh j-MYJOBFLOWID

hadoop@ip-10-160-75-214:~$ gdb `which python` /tmp/cores/core.python.blah
```

If you have multiple nodes, you may have to **scp** your identity file to the master node and use it to SSH to the slave nodes, where the core dumps are located:

```
hadoop@ip-10-160-75-214:~$ hadoop dfsadmin -report | grep ^Name
Name: 10.166.50.85:9200
Name: 10.177.63.114:9200

hadoop@ip-10-160-75-214:~$ ssh -i uploaded_key.pem 10.166.50.85

hadoop@ip-10-166-50-85:~$ gdb `which python2.6-dbg` /tmp/cores/core.python.blah
```

1.13.6 The `mrjob` command

The `mrjob` command has two purposes:

1. To provide easy access to EMR tools
2. To eventually let you run Hadoop Streaming jobs written in languages other than Python

EMR tools

`mrjob audit-emr-usage [options]` Audit EMR usage over the past 2 weeks, sorted by job flow name and user.

Alias for `mrjob.tools.emr.audit_usage`.

`mrjob create-job-flow [options]` Create a persistent EMR job flow, using bootstrap scripts and other configs from `mrjob.conf`, and print the job flow ID to stdout.

Alias for `mrjob.tools.emr.create_job_flow`.

`mrjob fetch-logs (job flow ID) [options]` List, display, and parse Hadoop logs associated with EMR job flows. Useful for debugging failed jobs for which mrjob did not display a useful error message or for inspecting jobs whose output has been lost.

Alias for `mrjob.tools.emr.fetch_logs`.

`mrjob report-long-jobs [options]` Report jobs running for more than a certain number of hours (by default, 24.0). This can help catch buggy jobs and Hadoop/EMR operational issues.

Alias for `mrjob.tools.emr.report_long_jobs`.

`mrjob s3-tmpwatch [options]` Delete all files in a given URI that are older than a specified time.

Alias for `mrjob.tools.emr.s3_tmpwatch`.

`mrjob terminate-idle-job-flows [options]` Terminate idle EMR job flows that meet the criteria passed in on the command line (or, by default, job flows that have been idle for one hour).

Alias for `mrjob.tools.emr.terminate_idle_job_flows`.

`mrjob terminate-job-flow (job flow ID)` Terminate an existing EMR job flow.

Alias for `mrjob.tools.emr.terminate_job_flow`.

Running jobs

`mrjob run (path to script or executable) [options]` Run a job. Takes same options as invoking a Python job. See *Options available to all runners*, *Hadoop-related options*, and *EMR runner options*. While you can use this command to invoke your jobs, you can just as easily call `python my_job.py [options]`.

1.13.7 EMR job flow management tools

Each tool can be invoked two ways: from the *mrjob command*, or by running the Python module directly. Both ways are given in each example.

audit_usage

Audit EMR usage over the past 2 weeks, sorted by job flow name and user.

Usage:

```
mrjob audit-emr-usage > report
python -m mrjob.tools.emr.audit_usage > report
```

Options:

```
-h, --help            show this help message and exit
-v, --verbose         print more messages to stderr
-q, --quiet           Don't log status messages; just print the report.
-c CONF_PATH, --conf-path=CONF_PATH
                      Path to alternate mrjob.conf file to read from
--no-conf             Don't load mrjob.conf even if it's available
--max-days-ago=MAX_DAYS_AGO
                      Max number of days ago to look at jobs. By default, we
                      go back as far as EMR supports (currently about 2
                      months)
```

create_job_flow

Create a persistent EMR job flow, using bootstrap scripts and other configs from `mrjob.conf`, and print the job flow ID to stdout.

Usage:

```
mrjob create-job-flow
python -m mrjob.tools.emr.create_job_flow
```

WARNING: do not run this without having `mrjob.tools.emr.terminate_idle_job_flows` in your crontab; job flows left idle can quickly become expensive!

fetch_logs

List, display, and parse Hadoop logs associated with EMR job flows. Useful for debugging failed jobs for which mrjob did not display a useful error message or for inspecting jobs whose output has been lost.

Usage:

```
mrjob fetch-logs -[l|L|a|A|--counters] [-s STEP_NUM] JOB_FLOW_ID
python -m mrjob.tools.emr.fetch_logs -[l|L|a|A|--counters] [-s STEP_NUM] JOB_FLOW_ID
```

Options:

```
-a, --cat             Cat log files MRJob finds relevant
-A, --cat-all         Cat all log files to JOB_FLOW_ID/
-c CONF_PATH, --conf-path=CONF_PATH
                      Path to alternate mrjob.conf file to read from
--counters            Show counters from the job flow
--ec2-key-pair-file=EC2_KEY_PAIR_FILE
                      Path to file containing SSH key for EMR
-h, --help            show this help message and exit
-l, --list            List log files MRJob finds relevant
-L, --list-all        List all log files
--no-conf             Don't load mrjob.conf even if it's available
```

```
-q, --quiet           Don't print anything to stderr
-s STEP_NUM, --step-num=STEP_NUM
                      Limit results to a single step. To be used with --list
                      and --cat.
-v, --verbose         print more messages to stderr
```

mrboss

Run a command on the master and all slaves. Store stdout and stderr for results in OUTPUT_DIR.

Usage:

```
python -m mrjob.tools.emr.mrboss JOB_FLOW_ID [options] "command string"
```

Options:

```
-c CONF_PATH, --conf-path=CONF_PATH
--ec2-key-pair-file=EC2_KEY_PAIR_FILE
                      Path to file containing SSH key for EMR
-h, --help            show this help message and exit
--no-conf             Don't load mrjob.conf even if it's available
-o, --output-dir      Specify an output directory (default: JOB_FLOW_ID)
-q, --quiet           Don't print anything to stderr
-v, --verbose         print more messages to stderr
```

report_long_jobs

Report jobs running for more than a certain number of hours (by default, 24.0). This can help catch buggy jobs and Hadoop/EMR operational issues.

Suggested usage: run this as a daily cron job with the `-q` option:

```
0 0 * * * mrjob report-long-jobs
0 0 * * * python -m mrjob.tools.emr.report_long_jobs -q
```

Options:

```
-h, --help            show this help message and exit
-v, --verbose         print more messages to stderr
-q, --quiet           Don't log status messages; just print the report.
-c CONF_PATH, --conf-path=CONF_PATH
                      Path to alternate mrjob.conf file to read from
--no-conf             Don't load mrjob.conf even if it's available
--min-hours=MIN_HOURS
                      Minimum number of hours a job can run before we report
                      it. Default: 24.0
```

s3_tmpwatch

Delete all files in a given URI that are older than a specified time. The time parameter defines the threshold for removing files. If the file has not been accessed for *time*, the file is removed. The time argument is a number with an optional single-character suffix specifying the units: m for minutes, h for hours, d for days. If no suffix is specified, time is in hours.

Suggested usage: run this as a cron job with the -q option:

```
0 0 * * * mrjob s3-tmpwatch -q 30d s3://your-bucket/tmp/
0 0 * * * python -m mrjob.tools.emr.s3_tmpwatch -q 30d s3://your-bucket/tmp/
```

Usage:

```
mrjob s3-tmpwatch [options] <time-untouched> <URIs>
python -m mrjob.tools.emr.s3_tmpwatch [options] <time-untouched> <URIs>
```

Options:

```
-h, --help            show this help message and exit
-v, --verbose         Print more messages
-q, --quiet           Report only fatal errors.
-c CONF_PATH, --conf-path=CONF_PATH
                      Path to alternate mrjob.conf file to read from
--no-conf             Don't load mrjob.conf even if it's available
-t, --test            Don't actually delete any files; just log that we
                      would
```

terminate_job_flow

Terminate an existing EMR job flow.

Usage:

```
mrjob terminate-job-flow [options] j-JOBFLOWID
python -m mrjob.tools.emr.terminate_job_flow [options] j-JOBFLOWID
```

Terminate an existing EMR job flow.

Options:

```
-h, --help            show this help message and exit
-v, --verbose         print more messages to stderr
-q, --quiet           don't print anything
-c CONF_PATH, --conf-path=CONF_PATH
                      Path to alternate mrjob.conf file to read from
--no-conf             Don't load mrjob.conf even if it's available
```

terminate_idle_job_flows

Terminate idle EMR job flows that meet the criteria passed in on the command line (or, by default, job flows that have been idle for one hour).

Suggested usage: run this as a cron job with the `-q` option:

```
*/30 * * * * mrjob terminate-idle-job-flows -q
*/30 * * * * python -m mrjob.tools.emr.terminate_idle_job_flows -q
```

Options:

```
-h, --help            show this help message and exit
-v, --verbose         Print more messages
-q, --quiet           Don't print anything to stderr; just print IDs of
                      terminated job flows and idle time information to
                      stdout. Use twice to print absolutely nothing.
-c CONF_PATH, --conf-path=CONF_PATH
                      Path to alternate mrjob.conf file to read from
```

```
--no-conf             Don't load mrjob.conf even if it's available
--max-hours-idle=MAX_HOURS_IDLE
                      Max number of hours a job flow can go without
                      bootstrapping, running a step, or having a new step
                      created. This will fire even if there are pending
                      steps which EMR has failed to start. Make sure you set
                      this higher than the amount of time your jobs can take
                      to start instances and bootstrap.
--mins-to-end-of-hour=MINS_TO_END_OF_HOUR
                      Terminate job flows that are within this many minutes
                      of the end of a full hour since the job started
                      running AND have no pending steps.
--unpooled-only       Only terminate un-pooled job flows
--pooled-only         Only terminate pooled job flows
--pool-name=POOL_NAME
                      Only terminate job flows in the given named pool.
--dry-run             Don't actually kill idle jobs; just log that we would
```

1.14 Contributing to mrjob

1.14.1 Contribution guidelines

mrjob is developed using a standard Github pull request process. Almost all code is reviewed in pull requests.

The general process for working on mrjob is:

- Fork the project on Github
- Clone your fork to your local machine
- Create a feature branch from master (e.g. `git branch delete_all_the_code`)
- Write code, commit often
- Write test cases for all changed functionality
- Submit a pull request against `master` on Github
- Wait for code review!

It would also help to discuss your ideas on the mailing list so we can warn you of possible merge conflicts with ongoing work or offer suggestions for where to put code.

Things that will make your branch more likely to be pulled:

- Comprehensive, fast test cases
- Detailed explanation of what the change is and how it works
- Reference relevant issue numbers in the tracker
- API backward compatibility

If you add a new configuration option, please try to do all of these things:

- Make its name unambiguous in the context of multiple runners (e.g. `ec2_task_instance_type` instead of `instance_type`)
- Add command line switches that allow full control over the option
- Document the option and its switches in the appropriate file under `docs`

1.14.2 A quick tour through the code

mrjob's modules can be put in four categories:

- Reading command line arguments and config files, and invoking machinery accordingly
  - `mrjob.conf`: Read config files
  - `mrjob.launch`: Invoke runners based on command line and configs
  - `mrjob.options`: Define command line options
- Interacting with Hadoop Streaming
  - `mrjob.job`: Python interface for writing jobs
  - `mrjob.protocol`: Defining data formats between Python steps
- Runners and support; submitting the job to various MapReduce environments
  - `mrjob.runner`: Common functionality across runners
  - `mrjob.hadoop`: Submit jobs to Hadoop
  - `mrjob.step`: Define/implement interface between runners and script steps
  - Local
    - `mrjob.inline`: Run Python-only jobs in-process
    - `mrjob.local`: Run Hadoop Streaming-only jobs in subprocesses
  - Amazon Elastic MapReduce
    - `mrjob.emr`: Submit jobs to EMR
    - `mrjob.pool`: Utilities for job flow pooling functionality
    - `mrjob.retry`: Wrapper for S3 and EMR connections to handle recoverable errors
    - `mrjob.ssh`: Run commands on EMR cluster machines
- Interacting with different "filesystems"
  - `mrjob.fs.base`: Common functionality
  - `mrjob.fs.composite`: Support multiple filesystems; if one fails, "fall through" to another
  - `mrjob.fs.hadoop`: HDFS
  - `mrjob.fs.local`: Local filesystem
  - `mrjob.fs.s3`: S3
  - `mrjob.fs.ssh`: SSH
- Utilities
  - `mrjob.compat`: Transparently handle differences between Hadoop versions
  - `mrjob.logparsers`: Find interesting information (errors, counters) in Hadoop logs (used by `hadoop` and `emr` runners)
  - `mrjob.parse`: Parsing utilities for URIs, logs, command line options, etc.
  - `mrjob.util`: Utilities for dealing with files, command line options, various other things

1.15 Interactions between runner and job

Warning: This information is **experimentally public** and subject to change.

Starting with version 0.4, mrjob is moving toward supporting arbitrary scripting languages for writing jobs. Jobs that don't use the `MRJob` Python class will need to support a simple interface for informing the runner of their steps and running the correct steps.

In this document, the **job script** is a file invoked with `<interpreter> script.blah`, which supports the interface described in this document and contains all information about how to run the job. In the normal case, the job script will be a file containing a single `MRJob` class and invocation, and `<interpreter` will be `python`.

All interactions between job and runner are through command line arguments. For example, to find out what mappers, reducers, and combiners a job has and what their order is, `MRJobRunner` calls the job script with the *`--steps`* argument.

Examples of job input/output are given at the end of this document in *Examples*.

1.15.1 Job Interface

`--steps` Print a JSON-encoded dictionary in the format described in *Format of –steps* describing the individual steps of the job.

`--step-num` Specify the step number to be run. **Always** used with *`--mapper`*, *`--combiner`*, or *`--reducer`*.

`--mapper` Run the mapper for the specified step. Always used with *`--step-num`*.

`--combiner` Run the combiner for the specified step. Always used with *`--step-num`*.

`--reducer` Run the reducer for the specified step. Always used with *`--step-num`*.

`--step-num`, *`--mapper`*, *`--combiner`*, and *`--reducer`* are only necessary for `script` steps (see *Format of –steps* below).

When running a mapper, combiner, or reducer, the non-option arguments are input files, where no args or `-` means read from standard input.

1.15.2 Format of `--steps`

Jobs are divided into **steps** which can either be a `jar` step or a `streaming` step.

Streaming steps

A `streaming` step consists of one or more **substeps** of type `mapper`, `combiner`, or `reducer`. Each substep can have type `script` or `command`. A `script` step follows the *`--step-num`* / *`--mapper`* / *`--combiner`* / *`--reducer`* interface, and a `command` is a raw command passed to Hadoop Streaming.

Script substeps

Here is a one-step streaming job with only a mapper in script format:

```
{
    'type': 'streaming',
    'mapper': {
        'type': 'script',
```

```
    }
}
```

Some Python code that would cause `MRJob` generate this data:

```
class MRMapperJob(MRJob):

    def steps(self):
        return [MRStep(mapper=self.my_mapper)]
```

The runners would then invoke Hadoop Streaming with:

```
-mapper 'mapper_job.py --mapper --step-num=0'
```

Script steps may have **pre-filters**, which are just UNIX commands that sit in front of the script when running the step, used to efficiently filter output with `grep` or otherwise filter and transform data. Filters are specified using a `pre_filter` key in the substep dictionary:

```
{
    'type': 'streaming',
    'mapper': {
        'type': 'script',
        'pre_filter': 'grep "specific data"'
    }
}
```

`MRJob` code:

```
class MRMapperFilterJob(MRJob):

    def steps(self):
        return [MRStep(mapper=self.my_mapper,
                       mapper_pre_filter='grep "specific data"')]
```

Hadoop Streaming arguments:

```
-mapper 'bash -c '\''grep "specific data" | mapper_job.py --mapper --step-num=0'\'''
```

mrjob does not try to intelligently handle quotes in the contents of filters, so avoid using single quotes.

Hadoop Streaming requires that all steps have a mapper, so if the job doesn't specify a mapper, mrjob will use `cat`.

Command substeps

The format for a command substep is very simple.

```
{
    'type': 'streaming',
    'mapper': {
        'type': 'command',
        'command': 'cat'
    }
}
```

`MRJob` code:

```
class MRMapperCommandJob(MRJob):

    def steps(self):
        return [MRStep(mapper_cmd='cat')]
```

Hadoop Streaming arguments:

```
-mapper 'cat'
```

Jar steps

Jar steps are used to specify jars that are not Hadoop Streaming. They have two required arguments and two optional arguments.

```
{
    'name': 'step_name',
    'jar': 'binks.jar.jar',
    'main_class': 'MyMainMan',      # optional
    'step_args': ['argh', 'argh']   # optional
}
```

Further information on jar steps should be sought for in the Hadoop documentation. Pull requests containing relevant links would be appreciated.

Examples

Getting steps

Job with a script mapper and command reducer for the first step and a jar for the second step:

```
> <interpreter> my_script.lang --steps
[
    {
        'type': 'streaming',
        'mapper': {
            'type': 'script'
        },
        'reducer': {
            'type': 'command',
            'command': 'some_shell_command --arg --arg'
        }
    },
    {
        'type': 'jar',
        'name': 'my_cool_jar_step',
        'jar': 's3://bucket/jar_jar.jar'
    }
]
```

Running a step

```
> <interpreter> my_script.lang --mapper --step-num=0 input.txt -
[script iterates over stdin and input.txt]
key_1       value_1
key_2       value_2
...
```

CHAPTER 2

Reference

2.1 mrjob.compat - Hadoop version compatibility

Utility functions for compatibility with different version of hadoop.

mrjob.compat.**add_translated_jobconf_for_hadoop_version**(*jobconf*, *hadoop_version*)
Translates the configuration property name to match those that are accepted in hadoop_version. Prints a warning message if any configuration property name does not match the name in the hadoop version. Combines the original jobconf with the translated jobconf.

Returns a map consisting of the original and translated configuration property names and values.

mrjob.compat.**get_jobconf_value**(*variable*, *default=None*)
Get the value of a jobconf variable from the runtime environment.

For example, a MRJob could use `jobconf_from_env('map.input.file')` to get the name of the file a mapper is reading input from.

If the name of the jobconf variable is different in different versions of Hadoop (e.g. in Hadoop 0.21, `map.input.file` is `mapreduce.map.input.file`), we'll automatically try all variants before giving up.

Return *default* if that jobconf variable isn't set.

mrjob.compat.**jobconf_from_dict**(*jobconf*, *name*, *default=None*)
Get the value of a jobconf variable from the given dictionary.

Parameters

- **jobconf** (*dict*) – jobconf dictionary
- **name** (*string*) – name of the jobconf variable (e.g. `'user.name'`)
- **default** – fallback value

If the name of the jobconf variable is different in different versions of Hadoop (e.g. in Hadoop 0.21, `map.input.file` is `mapreduce.map.input.file`), we'll automatically try all variants before giving up.

Return *default* if that jobconf variable isn't set.

mrjob.compat.**jobconf_from_env**(*variable*, *default=None*)
Get the value of a jobconf variable from the runtime environment.

For example, a MRJob could use `jobconf_from_env('map.input.file')` to get the name of the file a mapper is reading input from.

If the name of the jobconf variable is different in different versions of Hadoop (e.g. in Hadoop 0.21, `map.input.file` is `mapreduce.map.input.file`), we'll automatically try all variants before giving up.

Return *default* if that jobconf variable isn't set.

`mrjob.compat.`**`supports_combiners_in_hadoop_streaming`**(*version*)
Return `True` if this version of Hadoop Streaming supports combiners (i.e. >= 0.20.203), otherwise False.

`mrjob.compat.`**`supports_new_distributed_cache_options`**(*version*)
Use `-files` and `-archives` instead of `-cacheFile` and `-cacheArchive`

`mrjob.compat.`**`translate_jobconf`**(*variable*, *version*)
Translate *variable* to Hadoop version *version*. If it's not a variable we recognize, leave as-is.

`mrjob.compat.`**`uses_generic_jobconf`**(*version*)
Use `-D` instead of `-jobconf`

`mrjob.compat.`**`version_gte`**(*version*, *cmp_version_str*)
Return `True` if version >= *cmp_version_str*.

2.2 mrjob.conf - parse and write config files

"mrjob.conf" is the name of both this module, and the global config file for `mrjob`.

2.2.1 Reading and writing mrjob.conf

`mrjob.conf.`**`find_mrjob_conf`**()
Look for `mrjob.conf`, and return its path. Places we look:

- The location specified by `MRJOB_CONF`
- `~/.mrjob.conf`
- `/etc/mrjob.conf`

Return `None` if we can't find it.

`mrjob.conf.`**`load_opts_from_mrjob_conf`**(*runner_alias*, *conf_path=None*, *already_loaded=None*)
Load a list of dictionaries representing the options in a given mrjob.conf for a specific runner. Returns `[(path, values)]`. If conf_path is not found, return [(None, {})].

Parameters

- **runner_alias** (*str*) – String identifier of the runner type, e.g. `emr`, `local`, etc.
- **conf_path** (*str*) – location of the file to load
- **already_loaded** (*list*) – list of `mrjob.conf` paths that have already been loaded

`mrjob.conf.`**`load_opts_from_mrjob_confs`**(*runner_alias*, *conf_paths=None*)
Load a list of dictionaries representing the options in a given list of mrjob config files for a specific runner. Returns `[(path, values)]`. If a path is not found, use (None, {}) as its value. If *conf_paths* is `None`, look for a config file in the default locations.

Parameters

- **runner_alias** (*str*) – String identifier of the runner type, e.g. `emr`, `local`, etc.
- **conf_path** – locations of the files to load

2.2.2 Combining options

Combiner functions take a list of values to combine, with later options taking precedence over earlier ones. `None` values are always ignored.

`mrjob.conf.`**`combine_values`**`(*values)`
Return the last value in *values* that is not `None`.

The default combiner; good for simple values (booleans, strings, numbers).

`mrjob.conf.`**`combine_lists`**`(*seqs)`
Concatenate the given sequences into a list. Ignore `None` values.

Generally this is used for a list of commands we want to run; the "default" commands get run before any commands specific to your job.

`mrjob.conf.`**`combine_dicts`**`(*dicts)`
Combine zero or more dictionaries. Values from dicts later in the list take precedence over values earlier in the list.

If you pass in `None` in place of a dictionary, it will be ignored.

`mrjob.conf.`**`combine_cmds`**`(*cmds)`
Take zero or more commands to run on the command line, and return the last one that is not `None`. Each command should either be a list containing the command plus switches, or a string, which will be parsed with `shlex.split()`. The string must either be a byte string or a unicode string containing no non-ASCII characters.

Returns either `None` or a list containing the command plus arguments.

`mrjob.conf.`**`combine_cmd_lists`**`(*seqs_of_cmds)`
Concatenate the given commands into a list. Ignore `None` values, and parse strings with `shlex.split()`.

Returns a list of lists (each sublist contains the command plus arguments).

`mrjob.conf.`**`combine_envs`**`(*envs)`
Combine zero or more dictionaries containing environment variables.

Environment variables later from dictionaries later in the list take priority over those earlier in the list. For variables ending with `PATH`, we prepend (and add a colon) rather than overwriting.

If you pass in `None` in place of a dictionary, it will be ignored.

`mrjob.conf.`**`combine_local_envs`**`(*envs)`
Same as `combine_envs()`, except that paths are combined using the local path separator (e.g `;` on Windows rather than `:`).

`mrjob.conf.`**`combine_paths`**`(*paths)`
Returns the last value in *paths* that is not `None`. Resolve ~ (home dir) and environment variables.

`mrjob.conf.`**`combine_path_lists`**`(*path_seqs)`
Concatenate the given sequences into a list. Ignore None values. Resolve ~ (home dir) and environment variables, and expand globs that refer to the local filesystem.

2.3 mrjob.emr - run on EMR

2.3.1 Job Runner

class `mrjob.emr.`**`EMRJobRunner`**`(**kwargs)`
Runs an `MRJob` on Amazon Elastic MapReduce. Invoked when you run your job with `-r emr`.

EMRJobRunner runs your job in an EMR job flow, which is basically a temporary Hadoop cluster. Normally, it creates a job flow just for your job; it's also possible to run your job in a specific job flow by setting *emr_job_flow_id* or to automatically choose a waiting job flow, creating one if none exists, by setting *pool_emr_job_flows*.

Input, support, and jar files can be either local or on S3; use `s3://...` URLs to refer to files on S3.

This class has some useful utilities for talking directly to S3 and EMR, so you may find it useful to instantiate it without a script:

```
from mrjob.emr import EMRJobRunner

emr_conn = EMRJobRunner().make_emr_conn()
job_flows = emr_conn.describe_jobflows()
...
```

2.3.2 EMR Utilities

EMRJobRunner.**make_emr_conn**()
Create a connection to EMR.

Returns a boto.emr.connection.EmrConnection, wrapped in a mrjob.retry.RetryWrapper

mrjob.emr.**describe_all_job_flows**(*emr_conn*, *states=None*, *jobflow_ids=None*, *created_after=None*, *created_before=None*)
Iteratively call EmrConnection.describe_job_flows() until we really get all the available job flow information. Currently, 2 months of data is available through the EMR API.

This is a way of getting around the limits of the API, both on number of job flows returned, and how far back in time we can go.

Parameters

- **states** (*list*) – A list of strings with job flow states wanted
- **jobflow_ids** (*list*) – A list of job flow IDs
- **created_after** (*datetime*) – Bound on job flow creation time
- **created_before** (*datetime*) – Bound on job flow creation time

2.3.3 S3 Utilities

mrjob.fs.s3.**s3_key_to_uri**(*s3_key*)
Convert a boto Key object into an s3:// URI

class mrjob.emr.**S3Filesystem**(*aws_access_key_id*, *aws_secret_access_key*, *s3_endpoint*)
Filesystem for Amazon S3 URIs. Typically you will get one of these via EMRJobRunner().fs, composed with SSHFilesystem and LocalFilesystem.

S3Filesystem.**make_s3_conn**()
Create a connection to S3.

Returns a boto.s3.connection.S3Connection, wrapped in a mrjob.retry.RetryWrapper

S3Filesystem.**get_s3_key**(*uri*, *s3_conn=None*)
Get the boto Key object matching the given S3 uri, or return None if that key doesn't exist.

uri is an S3 URI: `s3://foo/bar`

You may optionally pass in an existing s3 connection through `s3_conn`.

`S3Filesystem.`**`get_s3_keys`**(*uri*, *s3_conn=None*)
Get a stream of boto Key objects for each key inside the given dir on S3.

uri is an S3 URI: `s3://foo/bar`

You may optionally pass in an existing S3 connection through s3_conn

`S3Filesystem.`**`get_s3_folder_keys`**(*uri*, *s3_conn=None*)
Deprecated since version 0.4.0.

Background: EMR used to fake directories on S3 by creating special `*_$folder$` keys in S3. That is no longer true, so this method is deprecated.

For example if your job outputs `s3://walrus/tmp/output/part-00000`, EMR will also create these keys:

- `s3://walrus/tmp_$folder$`
- `s3://walrus/tmp/output_$folder$`

If you want to grant another Amazon user access to your files so they can use them in S3, you must grant read access on the actual keys, plus any `*_$folder$` keys that "contain" your keys; otherwise EMR will error out with a permissions error.

This gets all the `*_$folder$` keys associated with the given URI, as boto Key objects.

This does not support globbing.

You may optionally pass in an existing S3 connection through `s3_conn`.

`S3Filesystem.`**`make_s3_key`**(*uri*, *s3_conn=None*)
Create the given S3 key, and return the corresponding boto Key object.

uri is an S3 URI: `s3://foo/bar`

You may optionally pass in an existing S3 connection through `s3_conn`.

2.4 mrjob.hadoop - run on your Hadoop cluster

class `mrjob.hadoop.`**`HadoopJobRunner`**(***kwargs*)
Runs an `MRJob` on your Hadoop cluster. Invoked when you run your job with `-r hadoop`.

Input and support files can be either local or on HDFS; use `hdfs://...` URLs to refer to files on HDFS.

`HadoopJobRunner.`**`__init__`**(***kwargs*)
`HadoopJobRunner` takes the same arguments as `MRJobRunner`, plus some additional options which can be defaulted in *mrjob.conf*.

2.4.1 Utilities

`mrjob.hadoop.`**`hadoop_log_dir`**(*hadoop_home=None*)
Return the path where Hadoop stores logs.

Parameters **hadoop_home** – putative value of `HADOOP_HOME`, or None to default to the actual value if used. This is only used if `HADOOP_LOG_DIR` is not defined.

mrjob.hadoop.**find_hadoop_streaming_jar**(*path*)
Return the path of the hadoop streaming jar inside the given directory tree, or None if we can't find it.

mrjob.hadoop.**fully_qualify_hdfs_path**(*path*)
If path isn't an hdfs:// URL, turn it into one.

2.5 mrjob.inline - debugger-friendly local testing

class mrjob.inline.**InlineMRJobRunner**(*mrjob_cls=None*, ***kwargs*)
Runs an MRJob in the same process, so it's easy to attach a debugger.

This is the default way to run jobs (we assume you'll spend some time debugging your job before you're ready to run it on EMR or Hadoop).

To more accurately simulate your environment prior to running on Hadoop/EMR, use `-r local` (see LocalMRJobRunner).

InlineMRJobRunner.**__init__**(*mrjob_cls=None*, ***kwargs*)
InlineMRJobRunner takes the same keyword args as MRJobRunner. However, please note:

- *hadoop_extra_args*, *hadoop_input_format*, *hadoop_output_format*, and *hadoop_streaming_jar*, and *partitioner* are ignored because they require Java. If you need to test these, consider starting up a standalone Hadoop instance and running your job with `-r hadoop`.
- *python_bin*, *setup*, *setup_cmds*, *setup_scripts* and *steps_python_bin* are ignored because we don't invoke subprocesses.

2.6 mrjob.job - defining your job

class mrjob.job.**MRJob**(*args=None*)
The base class for all MapReduce jobs. See `__init__()` for details.

2.6.1 One-step jobs

MRJob.**mapper**(*key*, *value*)
Re-define this to define the mapper for a one-step job.

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

Parameters

- **key** – A value parsed from input.
- **value** – A value parsed from input.

If you don't re-define this, your job will have a mapper that simply yields `(key, value)` as-is.

By default (if you don't mess with *Protocols*):

- `key` will be `None`
- `value` will be the raw input line, with newline stripped.
- `out_key` and `out_value` must be JSON-encodable: numeric, unicode, boolean, `None`, list, or dict whose keys are unicodes.

MRJob.**reducer**(*key*, *values*)
Re-define this to define the reducer for a one-step job.

Yields one or more tuples of `(out_key, out_value)`

Parameters

- **key** – A key which was yielded by the mapper
- **value** – A generator which yields all values yielded by the mapper which correspond to `key`.

By default (if you don't mess with *Protocols*):

- `out_key` and `out_value` must be JSON-encodable.
- `key` and `value` will have been decoded from JSON (so tuples will become lists).

MRJob.**combiner**(*key*, *values*)
Re-define this to define the combiner for a one-step job.

Yields one or more tuples of `(out_key, out_value)`

Parameters

- **key** – A key which was yielded by the mapper
- **value** – A generator which yields all values yielded by one mapper task/node which correspond to `key`.

By default (if you don't mess with *Protocols*):

- `out_key` and `out_value` must be JSON-encodable.
- `key` and `value` will have been decoded from JSON (so tuples will become lists).

MRJob.**mapper_init**()
Re-define this to define an action to run before the mapper processes any input.

One use for this function is to initialize mapper-specific helper structures.

Yields one or more tuples of `(out_key, out_value)`.

By default, `out_key` and `out_value` must be JSON-encodable; re-define `INTERNAL_PROTOCOL` to change this.

MRJob.**mapper_final**()
Re-define this to define an action to run after the mapper reaches the end of input.

One way to use this is to store a total in an instance variable, and output it after reading all input data. See `mrjob.examples` for an example.

Yields one or more tuples of `(out_key, out_value)`.

By default, `out_key` and `out_value` must be JSON-encodable; re-define `INTERNAL_PROTOCOL` to change this.

MRJob.**reducer_init**()
Re-define this to define an action to run before the reducer processes any input.

One use for this function is to initialize reducer-specific helper structures.

Yields one or more tuples of `(out_key, out_value)`.

By default, `out_key` and `out_value` must be JSON-encodable; re-define `INTERNAL_PROTOCOL` to change this.

`MRJob.`**`reducer_final`**`()`
Re-define this to define an action to run after the reducer reaches the end of input.

Yields one or more tuples of `(out_key, out_value)`.

By default, `out_key` and `out_value` must be JSON-encodable; re-define `INTERNAL_PROTOCOL` to change this.

`MRJob.`**`combiner_init`**`()`
Re-define this to define an action to run before the combiner processes any input.

One use for this function is to initialize combiner-specific helper structures.

Yields one or more tuples of `(out_key, out_value)`.

By default, `out_key` and `out_value` must be JSON-encodable; re-define `INTERNAL_PROTOCOL` to change this.

`MRJob.`**`combiner_final`**`()`
Re-define this to define an action to run after the combiner reaches the end of input.

Yields one or more tuples of `(out_key, out_value)`.

By default, `out_key` and `out_value` must be JSON-encodable; re-define `INTERNAL_PROTOCOL` to change this.

`MRJob.`**`mapper_cmd`**`()`
Re-define this to define the mapper for a one-step job **as a shell command.** If you define your mapper this way, the command will be passed unchanged to Hadoop Streaming, with some minor exceptions. For important specifics, see *Shell commands as steps*.

Basic example:

```
def mapper_cmd(self):
    return 'cat'
```

`MRJob.`**`reducer_cmd`**`()`
Re-define this to define the reducer for a one-step job **as a shell command.** If you define your mapper this way, the command will be passed unchanged to Hadoop Streaming, with some minor exceptions. For specifics, see *Shell commands as steps*.

Basic example:

```
def reducer_cmd(self):
    return 'cat'
```

`MRJob.`**`combiner_cmd`**`()`
Re-define this to define the combiner for a one-step job **as a shell command.** If you define your mapper this way, the command will be passed unchanged to Hadoop Streaming, with some minor exceptions. For specifics, see *Shell commands as steps*.

Basic example:

```
def combiner_cmd(self):
    return 'cat'
```

`MRJob.`**`mapper_pre_filter`**`()`
Re-define this to specify a shell command to filter the mapper's input before it gets to your job's mapper in a one-step job. For important specifics, see *Filtering task input with shell commands*.

Basic example:

```
def mapper_pre_filter(self):
    return 'grep "ponies"'
```

MRJob.**reducer_pre_filter**()
Re-define this to specify a shell command to filter the reducer's input before it gets to your job's reducer in a one-step job. For important specifics, see *Filtering task input with shell commands*.

Basic example:

```
def reducer_pre_filter(self):
    return 'grep "ponies"'
```

MRJob.**combiner_pre_filter**()
Re-define this to specify a shell command to filter the combiner's input before it gets to your job's combiner in a one-step job. For important specifics, see *Filtering task input with shell commands*.

Basic example:

```
def combiner_pre_filter(self):
    return 'grep "ponies"'
```

2.6.2 Multi-step jobs

MRJob.**steps**()
Re-define this to make a multi-step job.

If you don't re-define this, we'll automatically create a one-step job using any of `mapper()`, `mapper_init()`, `mapper_final()`, `reducer_init()`, `reducer_final()`, and `reducer()` that you've re-defined. For example:

```
def steps(self):
    return [self.mr(mapper=self.transform_input,
                    reducer=self.consolidate_1),
            self.mr(reducer_init=self.log_mapper_init,
                    reducer=self.consolidate_2)]
```

Returns a list of steps constructed with `mr()`

classmethod MRJob.**mr**(**args*, ***kwargs*)
A deprecated wrapper for `MRStep`, plus a little logic to support deprecated use of positional arguments.

classmethod MRJob.**jar**(**args*, ***kwargs*)
Alias for `JarStep`.

Deprecated since version 0.4.2.

2.6.3 Running the job

classmethod MRJob.**run**()
Entry point for running job from the command-line.

This is also the entry point when a mapper or reducer is run by Hadoop Streaming.

Does one of:

- Print step information (*--steps*). See `show_steps()`
- Run a mapper (*--mapper*). See `run_mapper()`

- Run a combiner (`--combiner`). See `run_combiner()`
- Run a reducer (`--reducer`). See `run_reducer()`
- Run the entire job. See `run_job()`

`MRJob.__init__`(*args=None*)

Entry point for running your job from other Python code.

You can pass in command-line arguments, and the job will act the same way it would if it were run from the command line. For example, to run your job on EMR:

```
mr_job = MRYourJob(args=['-r', 'emr'])
with mr_job.make_runner() as runner:
    ...
```

Passing in `None` is the same as passing in `[]` (if you want to parse args from `sys.argv`, call `MRJob.run()`).

For a full list of command-line arguments, run: `python -m mrjob.job --help`

`MRJob.make_runner`()

Make a runner based on command-line arguments, so we can launch this job on EMR, on Hadoop, or locally.

Return type `mrjob.runner.MRJobRunner`

2.6.4 Parsing output

`MRJob.parse_output_line`(*line*)

Parse a line from the final output of this MRJob into `(key, value)`. Used extensively in tests like this:

```
runner.run()
for line in runner.stream_output():
    key, value = mr_job.parse_output_line(line)
```

2.6.5 Counters and status messages

`MRJob.increment_counter`(*group*, *counter*, *amount=1*)

Increment a counter in Hadoop streaming by printing to stderr. If the type of either **group** or **counter** is `unicode`, then the counter will be written as unicode. Otherwise, the counter will be written as ASCII. Although writing non-ASCII will succeed, the resulting counter names may not be displayed correctly at the end of the job.

Parameters

- **group** (*str*) – counter group
- **counter** (*str*) – description of the counter
- **amount** (*int*) – how much to increment the counter by

Commas in `counter` or `group` will be automatically replaced with semicolons (commas confuse Hadoop streaming).

`MRJob.set_status`(*msg*)

Set the job status in hadoop streaming by printing to stderr.

This is also a good way of doing a keepalive for a job that goes a long time between outputs; Hadoop streaming usually times out jobs that give no output for longer than 10 minutes.

If the type of **msg** is `unicode`, then the message will be written as unicode. Otherwise, it will be written as ASCII.

2.6.6 Setting protocols

MRJob.**INPUT_PROTOCOL = <class 'mrjob.protocol.RawValueProtocol'>**
Protocol for reading input to the first mapper in your job. Default: `RawValueProtocol`.

For example you know your input data were in JSON format, you could set:

```
INPUT_PROTOCOL = JSONValueProtocol
```

in your class, and your initial mapper would receive decoded JSONs rather than strings.

See `mrjob.protocol` for the full list of protocols.

MRJob.**INTERNAL_PROTOCOL = <class 'mrjob.protocol.JSONProtocol'>**
Protocol for communication between steps and final output. Default: `JSONProtocol`.

For example if your step output weren't JSON-encodable, you could set:

```
INTERNAL_PROTOCOL = PickleProtocol
```

and step output would be encoded as string-escaped pickles.

See `mrjob.protocol` for the full list of protocols.

MRJob.**OUTPUT_PROTOCOL = <class 'mrjob.protocol.JSONProtocol'>**
Protocol to use for writing output. Default: `JSONProtocol`.

For example, if you wanted the final output in repr, you could set:

```
OUTPUT_PROTOCOL = ReprProtocol
```

See `mrjob.protocol` for the full list of protocols.

MRJob.**input_protocol**()
Instance of the protocol to use to convert input lines to Python objects. Default behavior is to return an instance of `INPUT_PROTOCOL`.

MRJob.**internal_protocol**()
Instance of the protocol to use to communicate between steps. Default behavior is to return an instance of `INTERNAL_PROTOCOL`.

MRJob.**output_protocol**()
Instance of the protocol to use to convert Python objects to output lines. Default behavior is to return an instance of `OUTPUT_PROTOCOL`.

MRJob.**pick_protocols**(*step_num*, *step_type*)
Pick the protocol classes to use for reading and writing for the given step.

Parameters

- **step_num** (*int*) – which step to run (e.g. `0` for the first step)
- **step_type** (*str*) – one of *'mapper'*, *'combiner'*, or *'reducer'*

Returns (read_function, write_function)

By default, we use one protocol for reading input, one internal protocol for communication between steps, and one protocol for final output (which is usually the same as the internal protocol). Protocols can be controlled by setting `INPUT_PROTOCOL`, `INTERNAL_PROTOCOL`, and `OUTPUT_PROTOCOL`.

Re-define this if you need fine control over which protocols are used by which steps.

2.6.7 Secondary sort

MRJob.**SORT_VALUES** = None

Set this to `True` if you would like reducers to receive the values associated with any key in sorted order (sorted by their *encoded* value). Also known as secondary sort.

This can be useful if you expect more values than you can fit in memory to be associated with one key, but you want to apply information in a small subset of these values to information in the other values. For example, you may want to convert counts to percentages, and to do this you first need to know the total count.

Even though values are sorted by their encoded value, most encodings will sort strings in order. For example, you could have values like: `['A', <total>]`, `['B', <count_name>, <count>]`, and the value containing the total should come first regardless of what protocol you're using.

See `jobconf()` and `partitioner()` for more about how this works.

New in version 0.4.1.

2.6.8 Command-line options

See *Defining command line options* for information on adding command line options to your job. See *Configuration quick reference* for a complete list of all configuration options.

MRJob.**configure_options**()

MRJob.**add_passthrough_option**(**args*, ***kwargs*)

Function to create options which both the job runner and the job itself respect (we use this for protocols, for example).

Use it like you would use `optparse.OptionParser.add_option()`:

```
def configure_options(self):
    super(MRYourJob, self).configure_options()
    self.add_passthrough_option(
        '--max-ngram-size', type='int', default=4, help='...')
```

Specify an *opt_group* keyword argument to add the option to that `OptionGroup` rather than the top-level `OptionParser`.

If you want to pass files through to the mapper/reducer, use `add_file_option()` instead.

MRJob.**add_file_option**(**args*, ***kwargs*)

Add a command-line option that sends an external file (e.g. a SQLite DB) to Hadoop:

```
def configure_options(self):
   super(MRYourJob, self).configure_options()
   self.add_file_option('--scoring-db', help=...)
```

This does the right thing: the file will be uploaded to the working dir of the script on Hadoop, and the script will be passed the same option, but with the local name of the file in the script's working directory.

We suggest against sending Berkeley DBs to your job, as Berkeley DB is not forwards-compatible (so a Berkeley DB that you construct on your computer may not be readable from within Hadoop). Use SQLite databases instead. If all you need is an on-disk hash table, try out the `sqlite3dbm` module.

MRJob.**load_options**(*args*)

Load command-line options into `self.options`.

Called from `__init__()` after `configure_options()`.

Parameters **args** (*list of str*) – a list of command line arguments. `None` will be treated the same as `[]`.

Re-define if you want to post-process command-line arguments:

```
def load_options(self, args):
    super(MRYourJob, self).load_options(args)

    self.stop_words = self.options.stop_words.split(',')
    ...
```

`MRJob.`**`is_mapper_or_reducer`**`()`
True if this is a mapper/reducer.

This is mostly useful inside `load_options()`, to disable loading options when we aren't running inside Hadoop Streaming.

`MRJob.`**`OPTION_CLASS = <class optparse.Option at 0x7f9162ab1e20>`**

2.6.9 Job runner configuration

`MRJob.`**`job_runner_kwargs`**`()`
Keyword arguments used to create runners when `make_runner()` is called.

Returns map from arg name to value

Re-define this if you want finer control of runner initialization.

You might find `mrjob.conf.combine_dicts()` useful if you want to add or change lots of keyword arguments.

`MRJob.`**`local_job_runner_kwargs`**`()`
Keyword arguments to create create runners when `make_runner()` is called, when we run a job locally (`-r local`).

Returns map from arg name to value

Re-define this if you want finer control when running jobs locally.

`MRJob.`**`emr_job_runner_kwargs`**`()`
Keyword arguments to create create runners when `make_runner()` is called, when we run a job on EMR (`-r emr`).

Returns map from arg name to value

Re-define this if you want finer control when running jobs on EMR.

`MRJob.`**`hadoop_job_runner_kwargs`**`()`
Keyword arguments to create create runners when `make_runner()` is called, when we run a job on EMR (`-r hadoop`).

Returns map from arg name to value

Re-define this if you want finer control when running jobs on hadoop.

`MRJob.`**`generate_passthrough_arguments`**`()`
Returns a list of arguments to pass to subprocesses, either on hadoop or executed via subprocess.

These are passed to `mrjob.runner.MRJobRunner.__init__()` as *extra_args*.

`MRJob.`**`generate_file_upload_args`**`()`
Figure out file upload args to pass through to the job runner.

Instead of generating a list of args, we're generating a list of tuples of `('--argname', path)`

These are passed to `mrjob.runner.MRJobRunner.__init__()` as `file_upload_args`.

classmethod `MRJob.`**`mr_job_script`**`()`
Path of this script. This returns the file containing this class.

2.6.10 Running specific parts of jobs

`MRJob.`**`run_job`**`()`
Run the all steps of the job, logging errors (and debugging output if `--verbose` is specified) to STDERR and streaming the output to STDOUT.

Called from `run()`. You'd probably only want to call this directly from automated tests.

`MRJob.`**`run_mapper`**`(`*step_num=0*`)`
Run the mapper and final mapper action for the given step.

Parameters **step_num** (*int*) – which step to run (0-indexed)

If we encounter a line that can't be decoded by our input protocol, or a tuple that can't be encoded by our output protocol, we'll increment a counter rather than raising an exception. If –strict-protocols is set, then an exception is raised

Called from `run()`. You'd probably only want to call this directly from automated tests.

`MRJob.`**`run_reducer`**`(`*step_num=0*`)`
Run the reducer for the given step.

Parameters **step_num** (*int*) – which step to run (0-indexed)

If we encounter a line that can't be decoded by our input protocol, or a tuple that can't be encoded by our output protocol, we'll increment a counter rather than raising an exception. If –strict-protocols is set, then an exception is raised

Called from `run()`. You'd probably only want to call this directly from automated tests.

`MRJob.`**`run_combiner`**`(`*step_num=0*`)`
Run the combiner for the given step.

Parameters **step_num** (*int*) – which step to run (0-indexed)

If we encounter a line that can't be decoded by our input protocol, or a tuple that can't be encoded by our output protocol, we'll increment a counter rather than raising an exception. If –strict-protocols is set, then an exception is raised

Called from `run()`. You'd probably only want to call this directly from automated tests.

`MRJob.`**`show_steps`**`()`
Print information about how many steps there are, and whether they contain a mapper or reducer. Job runners (see *Runners*) use this to determine how Hadoop should call this script.

Called from `run()`. You'd probably only want to call this directly from automated tests.

We currently output something like `MR M R`, but expect this to change!

2.6.11 Hadoop configuration

`MRJob.`**`HADOOP_INPUT_FORMAT`** **= None**
Optional name of an optional Hadoop `InputFormat` class, e.g. `'org.apache.hadoop.mapred.lib.NLineInputFormat'`.

Passed to Hadoop with the *first* step of this job with the `-inputformat` option.

If you require more sophisticated behavior, try `hadoop_input_format()` or the *hadoop_input_format* argument to `mrjob.runner.MRJobRunner.__init__()`.

`MRJob.`**`hadoop_input_format`**`()`
Optional Hadoop `InputFormat` class to parse input for the first step of the job.

Normally, setting `HADOOP_INPUT_FORMAT` is sufficient; redefining this method is only for when you want to get fancy.

`MRJob.`**`HADOOP_OUTPUT_FORMAT = None`**
Optional name of an optional Hadoop `OutputFormat` class, e.g. `'org.apache.hadoop.mapred.FileOutputFormat'`.

Passed to Hadoop with the *last* step of this job with the `-outputformat` option.

If you require more sophisticated behavior, try `hadoop_output_format()` or the *hadoop_output_format* argument to `mrjob.runner.MRJobRunner.__init__()`.

`MRJob.`**`hadoop_output_format`**`()`
Optional Hadoop `OutputFormat` class to write output for the last step of the job.

Normally, setting `HADOOP_OUTPUT_FORMAT` is sufficient; redefining this method is only for when you want to get fancy.

`MRJob.`**`JOBCONF = {}`**
Optional jobconf arguments we should always pass to Hadoop. This is a map from property name to value. e.g.:

`{'stream.num.map.output.key.fields': '4'}`

It's recommended that you only use this to hard-code things that affect the semantics of your job, and leave performance tweaks to the command line or whatever you use to launch your job.

`MRJob.`**`jobconf`**`()`
`-jobconf` args to pass to hadoop streaming. This should be a map from property name to value.

By default, this combines *jobconf* options from the command lines with `JOBCONF`, with command line arguments taking precedence.

If `SORT_VALUES` is set, we also set these jobconf values:

```
stream.num.map.output.key.fields=2
mapred.text.key.partitioner.options=k1,1
```

We also blank out `mapred.output.key.comparator.class` and `mapred.text.key.comparator.options` to prevent interference from `mrjob.conf`.

`SORT_VALUES` *can* be overridden by `JOBCONF`, the command line, and step-specific `jobconf` values.

For example, if you know your values are numbers, and want to sort them in reverse, you could do:

```
SORT_VALUES = True

JOBCONF = {
  'mapred.output.key.comparator.class':
      'org.apache.hadoop.mapred.lib.KeyFieldBasedComparator',
  'mapred.text.key.comparator.options': '-k1 -k2nr',
}
```

If you want to re-define this, it's strongly recommended that do something like this, so as not to inadvertently disable the *jobconf* option:

```
def jobconf(self):
    orig_jobconf = super(MyMRJobClass, self).jobconf()
    custom_jobconf = ...
```

```
        return mrjob.conf.combine_dicts(orig_jobconf, custom_jobconf)
```

MRJob.**PARTITIONER** = None
Optional Hadoop partitioner class to use to determine how mapper output should be sorted and distributed to reducers. For example: `'org.apache.hadoop.mapred.lib.HashPartitioner'`.

If you require more sophisticated behavior, try `partitioner()`.

MRJob.**partitioner**()
Optional Hadoop partitioner class to use to determine how mapper output should be sorted and distributed to reducers.

By default, returns whatever is passed to *--partitioner*, or if that option isn't used, `PARTITIONER`, or if that isn't set, and `SORT_VALUES` is true, it's set to `'org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner'`.

You probably don't need to re-define this; it's just here for completeness.

2.6.12 Hooks for testing

MRJob.**sandbox**(*stdin=None*, *stdout=None*, *stderr=None*)
Redirect stdin, stdout, and stderr for automated testing.

You can set stdin, stdout, and stderr to file objects. By default, they'll be set to empty `StringIO` objects. You can then access the job's file handles through `self.stdin`, `self.stdout`, and `self.stderr`. See *Testing jobs* for more information about testing.

You may call sandbox multiple times (this will essentially clear the file handles).

`stdin` is empty by default. You can set it to anything that yields lines:

```
mr_job.sandbox(stdin=StringIO('some_data\n'))
```

or, equivalently:

```
mr_job.sandbox(stdin=['some_data\n'])
```

For convenience, this sandbox() returns self, so you can do:

```
mr_job = MRJobClassToTest().sandbox()
```

Simple testing example:

```
mr_job = MRYourJob.sandbox()
self.assertEqual(list(mr_job.reducer('foo', ['a', 'b'])), [...])
```

More complex testing example:

```
from StringIO import StringIO

from mrjob.parse import parse_mr_job_stderr
from mrjob.protocol import JSONProtocol

mr_job = MRYourJob(args=[...])

fake_input = '"foo"\t"bar"\n"foo"\t"baz"\n'
mr_job.sandbox(stdin=StringIO(fake_input))

mr_job.run_reducer(link_num=0)
```

```
self.assertEqual(mrjob.stdout.getvalue(), ...)
self.assertEqual(parse_mr_job_stderr(mr_job.stderr), ...)
```

MRJob.**parse_output**(*protocol=None*)

Deprecated since version 0.4.2.

Parse the output from the given sandboxed job's `self.stdout`.

This was only useful for testing individual mappers/reducers without using a runner; normally you'd just use `runner.stream_output()`

Parameters **protocol** (*protocol*) – A protocol instance to use. Defaults to `JSONProtocol()`.

MRJob.**parse_counters**(*counters=None*)

Deprecated since version 0.4.2.

Parse the counters from the given sandboxed job's `self.stderr`; superseded `mrjob.parse.parse_mr_job_stderr()`.

This was only useful for testing individual mappers/reducers without a runner; normally you'd just use `runner.counters()`.

2.7 mrjob.local - simulate Hadoop locally with subprocesses

class mrjob.local.**LocalMRJobRunner**(***kwargs*)

Runs an MRJob locally, for testing purposes. Invoked when you run your job with `-r local`.

Unlike `InlineMRJobRunner`, this actually spawns multiple subprocesses for each task.

This is fairly inefficient and *not* a substitute for Hadoop; it's main purpose is to help you test out *setup* commands.

It's rare to need to instantiate this class directly (see `__init__()` for details).

LocalMRJobRunner.**__init__**(***kwargs*)

Arguments to this constructor may also appear in `mrjob.conf` under `runners/local`.

`LocalMRJobRunner`'s constructor takes the same keyword args as `MRJobRunner`. However, please note:

- *cmdenv* is combined with `combine_local_envs()`
- *python_bin* defaults to `sys.executable` (the current python interpreter)
- *hadoop_extra_args*, *hadoop_input_format*, *hadoop_output_format*, *hadoop_streaming_jar*, and *partitioner* are ignored because they require Java. If you need to test these, consider starting up a standalone Hadoop instance and running your job with `-r hadoop`.

2.8 mrjob.parse - log parsing

Utilities for parsing errors, counters, and status messages.

mrjob.parse.**counter_unescape**(*escaped_string*)

Fix names of counters and groups emitted by Hadoop 0.20+ logs, which use escape sequences for more characters than most decoders know about (e.g. `()`.).

Parameters **escaped_string** (*str*) – string from a counter log line

mrjob.parse.**find_hadoop_java_stack_trace**(*lines*)

Scan a log file or other iterable for a java stack trace from Hadoop, and return it as a list of lines.

In logs from EMR, we find java stack traces in `task-attempts/*/syslog`

Sample stack trace:

```
2010-07-27 18:25:48,397 WARN org.apache.hadoop.mapred.TaskTracker (main): Error running child
java.lang.OutOfMemoryError: Java heap space
        at org.apache.hadoop.mapred.IFile$Reader.readNextBlock(IFile.java:270)
        at org.apache.hadoop.mapred.IFile$Reader.next(IFile.java:332)
        at org.apache.hadoop.mapred.Merger$Segment.next(Merger.java:147)
        at org.apache.hadoop.mapred.Merger$MergeQueue.adjustPriorityQueue(Merger.java:238)
        at org.apache.hadoop.mapred.Merger$MergeQueue.next(Merger.java:255)
        at org.apache.hadoop.mapred.Merger.writeFile(Merger.java:86)
        at org.apache.hadoop.mapred.Merger$MergeQueue.merge(Merger.java:377)
        at org.apache.hadoop.mapred.Merger.merge(Merger.java:58)
        at org.apache.hadoop.mapred.ReduceTask.run(ReduceTask.java:277)
        at org.apache.hadoop.mapred.TaskTracker$Child.main(TaskTracker.java:2216)
```

(We omit the "Error running child" line from the results)

mrjob.parse.**find_input_uri_for_mapper**(*lines*)

Scan a log file or other iterable for the path of an input file for the first mapper on Hadoop. Just returns the path, or None if no match.

In logs from EMR, we find python tracebacks in `task-attempts/*/syslog`

Matching log lines look like:

```
2010-07-27 17:54:54,344 INFO org.apache.hadoop.fs.s3native.NativeS3FileSystem (main): Opening 's
```

mrjob.parse.**find_interesting_hadoop_streaming_error**(*lines*)

Scan a log file or other iterable for a hadoop streaming error other than "Job not Successful!". Return the error as a string, or None if nothing found.

In logs from EMR, we find java stack traces in `steps/*/syslog`

Example line:

```
2010-07-27 19:53:35,451 ERROR org.apache.hadoop.streaming.StreamJob (main): Error launching job
```

mrjob.parse.**find_job_log_multiline_error**(*lines*)

Scan a log file for an arbitrary multi-line error. Return it as a list of lines, or None of nothing was found.

Here is an example error:

```
MapAttempt TASK_TYPE="MAP" TASKID="task_201106280040_0001_m_000218" TASK_ATTEMPT_ID="attempt_201
java.io.IOException: Cannot run program "bash": java.io.IOException: error=12, Cannot allocate m
    at java.lang.ProcessBuilder.start(ProcessBuilder.java:460)
    at org.apache.hadoop.util.Shell.runCommand(Shell.java:149)
    at org.apache.hadoop.util.Shell.run(Shell.java:134)
    at org.apache.hadoop.fs.DF.getAvailable(DF.java:73)
    at org.apache.hadoop.fs.LocalDirAllocator$AllocatorPerContext.getLocalPathForWrite(LocalDirA
    at org.apache.hadoop.fs.LocalDirAllocator.getLocalPathForWrite(LocalDirAllocator.java:124)
    at org.apache.hadoop.mapred.TaskTracker.localizeJob(TaskTracker.java:648)
    at org.apache.hadoop.mapred.TaskTracker.startNewTask(TaskTracker.java:1320)
    at org.apache.hadoop.mapred.TaskTracker.offerService(TaskTracker.java:956)
    at org.apache.hadoop.mapred.TaskTracker.run(TaskTracker.java:1357)
    at org.apache.hadoop.mapred.TaskTracker.main(TaskTracker.java:2361)
Caused by: java.io.IOException: java.io.IOException: error=12, Cannot allocate memory
    at java.lang.UNIXProcess.<init>(UNIXProcess.java:148)
```

```
        at java.lang.ProcessImpl.start(ProcessImpl.java:65)
        at java.lang.ProcessBuilder.start(ProcessBuilder.java:453)
        ... 10 more
    "
```

The first line returned will only include the text after `ERROR="`, and discard the final line with just `"`.

These errors are parsed from jobs/*.jar.

`mrjob.parse.`**`find_python_traceback`**(*lines*)
Scan a log file or other iterable for a Python traceback, and return it as a list of lines.

In logs from EMR, we find python tracebacks in `task-attempts/*/stderr`

`mrjob.parse.`**`find_timeout_error`**(*lines*)
Scan a log file or other iterable for a timeout error from Hadoop. Return the number of seconds the job ran for before timing out, or None if nothing found.

In logs from EMR, we find timeouterrors in `jobs/*.jar`

Example line:

```
Task TASKID="task_201010202309_0001_m_000153" TASK_TYPE="MAP" TASK_STATUS="FAILED" FINISH_TIME="
```

`mrjob.parse.`**`is_s3_uri`**(*uri*)
Return True if *uri* can be parsed into an S3 URI, False otherwise.

`mrjob.parse.`**`is_uri`**(*uri*)
Return True if *uri* is any sort of URI.

`mrjob.parse.`**`is_windows_path`**(*uri*)
Return True if *uri* is a windows path.

`mrjob.parse.`**`parse_hadoop_counters_from_line`**(*line*, *hadoop_version=None*)
Parse Hadoop counter values from a log line.

The counter log line format changed significantly between Hadoop 0.18 and 0.20, so this function switches between parsers for them.

Parameters **line** (*str*) – log line containing counter data

Returns (counter_dict, step_num) or (None, None)

`mrjob.parse.`**`parse_key_value_list`**(*kv_string_list*, *error_fmt*, *error_func*)
Parse a list of strings like `KEY=VALUE` into a dictionary.

Parameters

- **kv_string_list** (*[str]*) – Parse a list of strings like `KEY=VALUE` into a dictionary.
- **error_fmt** (*str*) – Format string accepting one `%s` argument which is the malformed (i.e. not `KEY=VALUE`) string
- **error_func** (*function(str)*) – Function to call when a malformed string is encountered.

`mrjob.parse.`**`parse_mr_job_stderr`**(*stderr*, *counters=None*)
Parse counters and status messages out of MRJob output.

Parameters

- **stderr** – a filehandle, a list of lines, or a str containing data
- **counters** – Counters so far, to update; a map from group to counter name to count.

Returns a dictionary with the keys *counters*, *statuses*, *other*:

- *counters*: counters so far; same format as above
- *statuses*: a list of status messages encountered
- *other*: lines that aren't either counters or status messages

mrjob.parse.**parse_port_range_list**(*range_list_str*)
Parse a port range list of the form (start[:end])(,(start[:end]))*

mrjob.parse.**parse_s3_uri**(*uri*)
Parse an S3 URI into (bucket, key)

```
>>> parse_s3_uri('s3://walrus/tmp/')
('walrus', 'tmp/')
```

If `uri` is not an S3 URI, raise a ValueError

2.9 mrjob.protocol - input and output

Protocols deserialize and serialize the input and output of tasks to raw bytes for Hadoop to distribute to the next task or to write as output. For more information, see *Protocols* and *Writing custom protocols*.

class mrjob.protocol.**JSONProtocol**
Encode `(key, value)` as two JSONs separated by a tab.

Note that JSON has some limitations; dictionary keys must be strings, and there's no distinction between lists and tuples.

class mrjob.protocol.**JSONValueProtocol**
Encode `value` as a JSON and discard `key` (`key` is read in as `None`).

class mrjob.protocol.**PickleProtocol**
Encode `(key, value)` as two string-escaped pickles separated by a tab.

We string-escape the pickles to avoid having to deal with stray `\t` and `\n` characters, which would confuse Hadoop Streaming.

Ugly, but should work for any type.

class mrjob.protocol.**PickleValueProtocol**
Encode `value` as a string-escaped pickle and discard `key` (`key` is read in as `None`).

class mrjob.protocol.**RawProtocol**
Encode `(key, value)` as `key` and `value` separated by a tab (`key` and `value` should be bytestrings).

If `key` or `value` is `None`, don't include a tab. When decoding a line with no tab in it, `value` will be `None`.

When reading from a line with multiple tabs, we break on the first one.

Your key should probably not be `None` or have tab characters in it, but we don't check.

class mrjob.protocol.**RawValueProtocol**
Read in a line as `(None, line)`. Write out `(key, value)` as `value`. `value` must be a `str`.

The default way for a job to read its initial input.

class mrjob.protocol.**ReprProtocol**
Encode `(key, value)` as two reprs separated by a tab.

This only works for basic types (we use `mrjob.util.safeeval()`).

class mrjob.protocol.**ReprValueProtocol**
Encode `value` as a repr and discard `key` (`key` is read in as None).

This only works for basic types (we use `mrjob.util.safeeval()`).

2.10 mrjob.retry - retry on transient errors

class mrjob.retry.**RetryWrapper**(*wrapped*, *retry_if*, *backoff=15*, *multiplier=1.5*, *max_tries=10*)
Handle transient errors, with configurable backoff.

This class can wrap any object. The wrapped object will behave like the original one, except that if you call a function and it raises a retriable exception, we'll back off for a certain number of seconds and call the function again, until it succeeds or we get a non-retriable exception.

RetryWrapper.**__init__**(*wrapped*, *retry_if*, *backoff=15*, *multiplier=1.5*, *max_tries=10*)
Wrap the given object

Parameters

- **wrapped** – the object to wrap
- **retry_if** – a method that takes an exception, and returns whether we should retry
- **backoff** (*float*) – the number of seconds to wait the first time we get a retriable error
- **multiplier** (*float*) – if we retry multiple times, the amount to multiply the backoff time by every time we get an error
- **max_tries** (*int*) – how many tries we get. `0` means to keep trying forever

2.11 mrjob.runner - base class for all runners

class mrjob.runner.**MRJobRunner**(*mr_job_script=None*, *conf_path=None*, *extra_args=None*, *file_upload_args=None*, *hadoop_input_format=None*, *hadoop_output_format=None*, *input_paths=None*, *output_dir=None*, *partitioner=None*, *stdin=None*, *conf_paths=None*, ***opts*)
Abstract base class for all runners

MRJobRunner.**__init__**(*mr_job_script=None*, *conf_path=None*, *extra_args=None*, *file_upload_args=None*, *hadoop_input_format=None*, *hadoop_output_format=None*, *input_paths=None*, *output_dir=None*, *partitioner=None*, *stdin=None*, *conf_paths=None*, ***opts*)
All runners take the following keyword arguments:

Parameters

- **mr_job_script** (*str*) – the path of the `.py` file containing the `MRJob`. If this is None, you won't actually be able to `run()` the job, but other utilities (e.g. `ls()`) will work.
- **conf_path** (*str, None, or False*) – Deprecated. Alternate path to read configs from, or `False` to ignore all config files. Use *conf_paths* instead.
- **conf_paths** (*None or list*) – List of config files to combine and use, or None to search for mrjob.conf in the default locations.
- **extra_args** (*list of str*) – a list of extra cmd-line arguments to pass to the mr_job script. This is a hook to allow jobs to take additional arguments.

- **file_upload_args** – a list of tuples of `('--ARGNAME', path)`. The file at the given path will be uploaded to the local directory of the mr_job script when it runs, and then passed into the script with `--ARGNAME`. Useful for passing in SQLite DBs and other configuration files to your job.
- **hadoop_input_format** (*str*) – name of an optional Hadoop `InputFormat` class. Passed to Hadoop along with your first step with the `-inputformat` option. Note that if you write your own class, you'll need to include it in your own custom streaming jar (see *hadoop_streaming_jar*).
- **hadoop_output_format** (*str*) – name of an optional Hadoop `OutputFormat` class. Passed to Hadoop along with your first step with the `-outputformat` option. Note that if you write your own class, you'll need to include it in your own custom streaming jar (see *hadoop_streaming_jar*).
- **input_paths** (*list of str*) – Input files for your job. Supports globs and recursively walks directories (e.g. `['data/common/', 'data/training/*.gz']`). If this is left blank, we'll read from stdin
- **output_dir** (*str*) – An empty/non-existent directory where Hadoop streaming should put the final output from the job. If you don't specify an output directory, we'll output into a subdirectory of this job's temporary directory. You can control this from the command line with `--output-dir`. This option cannot be set from configuration files. If used with the hadoop runner, this path does not need to be fully qualified with `hdfs://` URIs because it's understood that it has to be on HDFS.
- **partitioner** (*str*) – Optional name of a Hadoop partitoner class, e.g. `'org.apache.hadoop.mapred.lib.HashPartitioner'`. Hadoop streaming will use this to determine how mapper output should be sorted and distributed to reducers.
- **stdin** – an iterable (can be a `StringIO` or even a list) to use as stdin. This is a hook for testing; if you set `stdin` via `sandbox()`, it'll get passed through to the runner. If for some reason your lines are missing newlines, we'll add them; this makes it easier to write automated tests.

2.11.1 Running your job

`MRJobRunner.`**`run`**`()`

Run the job, and block until it finishes.

Raise an exception if there are any problems.

`MRJobRunner.`**`stream_output`**`()`

Stream raw lines from the job's output. You can parse these using the read() method of the appropriate HadoopStreamingProtocol class.

`MRJobRunner.`**`cleanup`**`(`*mode=None*`)`

Clean up running jobs, scratch dirs, and logs, subject to the *cleanup* option passed to the constructor.

If you create your runner in a `with` block, `cleanup()` will be called automatically:

```
with mr_job.make_runner() as runner:
    ...

# cleanup() called automatically here
```

Parameters **mode** – override *cleanup* passed into the constructor. Should be a list of strings from CLEANUP_CHOICES

mrjob.runner.**CLEANUP_CHOICES = ['ALL', 'LOCAL_SCRATCH', 'LOGS', 'NONE', 'REMOTE_SCRATCH', 'SCRATC**
cleanup options:

- 'ALL': delete local scratch, remote scratch, and logs; stop job flow if on EMR and the job is not done when cleanup is run.
- 'LOCAL_SCRATCH': delete local scratch only
- 'LOGS': delete logs only
- 'NONE': delete nothing
- 'REMOTE_SCRATCH': delete remote scratch only
- 'SCRATCH': delete local and remote scratch, but not logs
- 'JOB': stop job if on EMR and the job is not done when cleanup runs
- **'JOB_FLOW': terminate the job flow if on EMR and the job is not done** on cleanup
- 'IF_SUCCESSFUL' (deprecated): same as ALL. Not supported for cleanup_on_failure.

2.11.2 Run Information

MRJobRunner.**counters**()
Get counters associated with this run in this form:

```
[{'group name': {'counter1': 1, 'counter2': 2}},
 {'group name': ...}]
```

The list contains an entry for every step of the current job, ignoring earlier steps in the same job flow.

MRJobRunner.**get_hadoop_version**()
Return the version number of the Hadoop environment as a string if Hadoop is being used or simulated. Return None if not applicable.

EMRJobRunner infers this from the job flow. HadoopJobRunner gets this from hadoop version. LocalMRJobRunner has an additional *hadoop_version* option to specify which version it simulates, with a default of 0.20. InlineMRJobRunner does not simulate Hadoop at all.

2.11.3 Configuration

MRJobRunner.**get_opts**()
Get options set for this runner, as a dict.

2.11.4 File management

MRJobRunner.**fs**
Filesystem object for the local filesystem. Methods on Filesystem objects will be forwarded to MRJobRunner until mrjob 0.5, but **this behavior is deprecated.**

class mrjob.fs.base.**Filesystem**
Some simple filesystem operations that are common across the local filesystem, S3, HDFS, and remote machines via SSH. Different runners provide functionality for different filesystems via their fs attribute. The hadoop and emr runners provide support for multiple protocols using CompositeFilesystem.

Protocol support:

- `mrjob.fs.hadoop.HadoopFilesystem`: `hdfs://`, others
- `mrjob.fs.local.LocalFilesystem`: `/`
- `mrjob.fs.s3.S3Filesystem`: `s3://bucket/path`, `s3n://bucket/path`
- `mrjob.fs.ssh.SSHFilesystem`: `ssh://hostname/path`

cat (*path_glob*)
cat all files matching **path_glob**, decompressing if necessary

du (*path_glob*)
Get the total size of files matching `path_glob`

Corresponds roughly to: `hadoop fs -dus path_glob`

ls (*path_glob*)
Recursively list all files in the given path.

We don't return directories for compatibility with S3 (which has no concept of them)

Corresponds roughly to: `hadoop fs -lsr path_glob`

md5sum (*path_glob*)
Generate the md5 sum of the file at `path`

mkdir (*path*)
Create the given dir and its subdirs (if they don't already exist).

Corresponds roughly to: `hadoop fs -mkdir path`

path_exists (*path_glob*)
Does the given path exist?

Corresponds roughly to: `hadoop fs -test -e path_glob`

rm (*path_glob*)
Recursively delete the given file/directory, if it exists

Corresponds roughly to: `hadoop fs -rmr path_glob`

touchz (*path*)
Make an empty file in the given location. Raises an error if a non-zero length file already exists in that location.

Correponds to: `hadoop fs -touchz path`

2.12 mrjob.step - represent Job Steps

class `mrjob.step.`**JarStep** (**args*, ***kwargs*)
Represents a running a custom Jar as a step.

INPUT = '<input>'
If this is passed as one of the step's arguments, it'll be replaced with the step's input paths (if there are multiple paths, they'll be joined with commas)

OUTPUT = '<output>'
If this is passed as one of the step's arguments, it'll be replaced with the step's output path

description (*step_num*)
Returns a dictionary representation of this step:

```
{
    'type': 'jar',
    'jar': path of the jar,
    'main_class': string, name of the main class,
    'args': list of strings, args to the main class,
}
```

See *Format of –steps* for examples.

mrjob.step.**MRJobStep**
alias of MRStep

class mrjob.step.**MRStep**(***kwargs*)
Represents steps handled by the script containing your job.

Used by MRJob.steps. See *Multi-step jobs* for sample usage.

Accepts the following keyword arguments.

Parameters

- **mapper** – function with same function signature as mapper(), or None for an identity mapper.
- **reducer** – function with same function signature as reducer(), or None for no reducer.
- **combiner** – function with same function signature as combiner(), or None for no combiner.
- **mapper_init** – function with same function signature as mapper_init(), or None for no initial mapper action.
- **mapper_final** – function with same function signature as mapper_final(), or None for no final mapper action.
- **reducer_init** – function with same function signature as reducer_init(), or None for no initial reducer action.
- **reducer_final** – function with same function signature as reducer_final(), or None for no final reducer action.
- **combiner_init** – function with same function signature as combiner_init(), or None for no initial combiner action.
- **combiner_final** – function with same function signature as combiner_final(), or None for no final combiner action.
- **jobconf** – dictionary with custom jobconf arguments to pass to hadoop.

description(*step_num*)
Returns a dictionary representation of this step:

```
{
    'type': 'streaming',
    'mapper': { ... },
    'combiner': { ... },
    'reducer': { ... },
    'jobconf': dictionary of Hadoop configuration properties
}
```

jobconf is optional, and only one of mapper, combiner, and reducer need be included.

mapper, combiner, and reducer are either handled by the script containing your job definition:

```
{
    'type': 'script',
    'pre_filter': (optional) cmd to pass input through, as a string
}
```

or they simply run a command:

```
{
    'type': 'command',
    'command': command to run, as a string
}
```

See *Format of –steps* for examples.

2.13 mrjob.setup - job environment setup

Utilities for setting up the environment jobs run in by uploading files and running setup scripts.

The general idea is to use Hadoop DistributedCache-like syntax to find and parse expressions like `/path/to/file#name_in_working_dir` into "path dictionaries" like `{'type': 'file', 'path': '/path/to/file', 'name': 'name_in_working_dir'}`.

You can then pass these into a `WorkingDirManager` to keep track of which files need to be uploaded, catch name collisions, and assign names to unnamed paths (e.g. `/path/to/file#`). Note that `WorkingDirManager.name()` can take a path dictionary as keyword arguments.

If you need to upload files from the local filesystem to a place where Hadoop can see them (HDFS or S3), we provide `UploadDirManager`.

Path dictionaries are meant to be immutable; all state is handled by manager classes.

class `mrjob.setup.`**`BootstrapWorkingDirManager`**
Manage the working dir for the master bootstrap script. Identical to `WorkingDirManager` except that it doesn't support archives.

class `mrjob.setup.`**`UploadDirManager`**(*prefix*)
Represents a directory on HDFS or S3 where we want to upload local files for consumption by Hadoop.

`UploadDirManager` tries to give files the same name as their filename in the path (for ease of debugging), but handles collisions gracefully.

`UploadDirManager` assumes URIs to not need to be uploaded and thus does not store them. `uri()` maps URIs to themselves.

`add`(*path*)

Add a path. If *path* hasn't been added before, assign it a name. If *path* is a URI don't add it; just return the URI.

Returns the URI assigned to the path

`path_to_uri`()
Get a map from path to URI for all paths that were added, so we can figure out which files we need to upload.

`uri`(*path*)
Get the URI for the given path. If *path* is a URI, just return it.

class mrjob.setup.**WorkingDirManager**
Represents the working directory of hadoop tasks (or bootstrap commands on EMR).

To support Hadoop's distributed cache, paths can be for ordinary files, or for archives (which are automatically uncompressed into a directory by Hadoop).

When adding a file, you may optionally assign it a name; if you don't; we'll lazily assign it a name as needed. Name collisions are not allowed, so being lazy makes it easier to avoid unintended collisions.

If you wish, you may assign multiple names to the same file, or add a path as both a file and an archive (though not mapped to the same name).

add(*type*, *path*, *name=None*)
Add a path as either a file or an archive, optionally assigning it a name.

Parameters

- **type** – either 'archive' or 'file'
- **path** – path/URI to add
- **name** – optional name that this path *must* be assigned, or None to assign this file a name later.

name(*type*, *path*, *name=None*)
Get the name for a path previously added to this WorkingDirManager, assigning one as needed.

This is primarily for getting the name of auto-named files. If the file was added with an assigned name, you must include it (and we'll just return *name*).

We won't ever give an auto-name that's the same an assigned name (even for the same path and type).

Parameters

- **type** – either 'archive' or 'file'
- **path** – path/URI
- **name** – known name of the file

name_to_path(*type*)
Get a map from name (in the setup directory) to path for all known files/archives, so we can build *-file* and *-archive* options to Hadoop (or fake them in a bootstrap script).

Parameters **type** – either 'archive' or 'file'

paths()
Get a set of all paths tracked by this WorkingDirManager.

mrjob.setup.**name_uniquely**(*path*, *names_taken=()*, *proposed_name=None*)
Come up with a unique name for *path*.

Parameters

- **names_taken** – a dictionary or set of names not to use.
- **proposed_name** – name to use if it is not taken. If this is not set, we propose a name based on the filename.

If the proposed name is taken, we add a number to the end of the filename, keeping the extension the same. For example:

```
>>> name_uniquely('foo.tar.gz', set(['foo.tar.gz']))
'foo-1.tar.gz'
```

mrjob.setup.**parse_legacy_hash_path**(*type*, *path*, *must_name=None*)
Parse hash paths from old setup/bootstrap options.

This is similar to parsing hash paths out of shell commands (see `parse_setup_cmd()`) except that we pass in path type explicitly, and we don't always require the `#` character.

Parameters

- **type** – Type of the path (`'archive'` or `'file'`)
- **path** – Path to parse, possibly with a `#`
- **must_name** – If set, use *path*'s filename as its name if there is no `'#'` in *path*, and raise an exception if there is just a `'#'` with no name. Set *must_name* to the name of the relevant option so we can print a useful error message. This is intended for options like `upload_files` that merely upload a file without tracking it.

mrjob.setup.**parse_setup_cmd**(*cmd*)
Parse a setup/bootstrap command, finding and pulling out Hadoop Distributed Cache-style paths ("hash paths").

Parameters **cmd** (*string*) – shell command to parse

Returns a list containing dictionaries (parsed hash paths) and strings (parts of the original command, left unparsed)

Hash paths look like `path#name`, where *path* is either a local path or a URI pointing to something we want to upload to Hadoop/EMR, and *name* is the name we want it to have when we upload it; *name* is optional (no name means to pick a unique one).

If *name* is followed by a trailing slash, that indicates *path* is an archive (e.g. a tarball), and should be unarchived into a directory on the remote system. The trailing slash will *also* be kept as part of the original command.

Parsed hash paths are dicitionaries with the keys `path`, `name`, and `type` (either `'file'` or `'archive'`).

Most of the time, this function will just do what you expect. Rules for finding hash paths:

- we only look for hash paths outside of quoted strings
- *path* may not contain `:` or `=` unless it is a URI (starts with `<scheme>://`); this allows you to do stuff like `export PYTHONPATH=$PYTHONPATH:foo.egg#`.
- *name* may not contain whitespace or any of the following characters: `'";;><|=/#`, so you can do stuff like `sudo dpkg -i fooify.deb#; fooify bar`

If you really want to include forbidden characters, you may use backslash escape sequences in *path* and *name*. (We can't guarantee Hadoop/EMR will accept them though!). Also, remember that shell syntax allows you to concatenate strings `like""this`.

Environment variables and `~` (home dir) in *path* will be resolved (use backslash escapes to stop this). We don't resolve *name* because it doesn't make sense. Environment variables and `~` elsewhere in the command are considered to be part of the script and will be resolved on the remote system.

2.14 mrjob.util - general utility functions

Utility functions for MRJob that have no external dependencies.

mrjob.util.**args_for_opt_dest_subset**(*option_parser*, *args*, *dests=None*)
For the given `OptionParser` and list of command line arguments *args*, yield values in *args* that correspond to option destinations in the set of strings *dests*. If *dests* is None, return *args* as parsed by `OptionParser`.

mrjob.util.**bash_wrap**(*cmd_str*)
Escape single quotes in a shell command string and wrap it with `bash -c '<string>'`.

This low-tech replacement works because we control the surrounding string and single quotes are the only character in a single-quote string that needs escaping.

mrjob.util.**buffer_iterator_to_line_iterator**(*iterator*)
boto's file iterator splits by buffer size instead of by newline. This wrapper puts them back into lines.

> **Warning:** This may append a newline to your last chunk of data. In v0.5.0 it will not, for better compatibility with file objects.

mrjob.util.**bunzip2_stream**(*fileobj*, *bufsize=1024*)
Decompress gzipped data on the fly.

Parameters

- **fileobj** – object supporting `read()`
- **bufsize** – number of bytes to read from *fileobj* at a time.

> **Warning:** This yields lines for backwards compatibility only; in v0.5.0 it will yield arbitrary chunks of data as part of supporting non-line-based protocols (see Issue #715). If you want lines, wrap this in `buffer_iterator_to_line_iterator()`.

mrjob.util.**cmd_line**(*args*)
build a command line that works in a shell.

mrjob.util.**expand_path**(*path*)
Resolve ~ (home dir) and environment variables in *path*.

If *path* is `None`, return `None`.

mrjob.util.**extract_dir_for_tar**(*archive_path*, *compression='gz'*)
Deprecated since version 0.4.0.

Get the name of the directory the tar at *archive_path* extracts into.

Parameters

- **archive_path** (*str*) – path to archive file
- **compression** (*str*) – Compression type to use. This can be one of `''`, `bz2`, or `gz`.

mrjob.util.**file_ext**(*path*)
return the file extension, including the `.`

```
>>> file_ext('foo.tar.gz')
'.tar.gz'
```

mrjob.util.**gunzip_stream**(*fileobj*, *bufsize=1024*)
Decompress gzipped data on the fly.

Parameters

- **fileobj** – object supporting `read()`
- **bufsize** – number of bytes to read from *fileobj* at a time. The default is the same as in `gzip`.

> **Warning:** This yields decompressed chunks; it does *not* split on lines. To get lines, wrap this in `buffer_iterator_to_line_iterator()`.

mrjob.util.**hash_object**(*obj*)
Generate a hash (currently md5) of the `repr` of the object

mrjob.util.**is_ironpython** = **False**
Deprecated since version 0.4.

mrjob.util.**log_to_null**(*name=None*)
Set up a null handler for the given stream, to suppress "no handlers could be found" warnings.

mrjob.util.**log_to_stream**(*name=None*, *stream=None*, *format=None*, *level=None*, *debug=False*)
Set up logging.

Parameters

- **name** (*str*) – name of the logger, or `None` for the root logger
- **stderr** (*file object*) – stream to log to (default is `sys.stderr`)
- **format** (*str*) – log message format (default is '%(message)s')
- **level** – log level to use
- **debug** (*bool*) – quick way of setting the log level: if true, use `logging.DEBUG`, otherwise use `logging.INFO`

mrjob.util.**parse_and_save_options**(*option_parser*, *args*)
DEPRECATED. To be removed in v0.5.

Duplicate behavior of `OptionParser`, but capture the strings required to reproduce the same values. Ref. optparse.py lines 1414-1548 (python 2.6.5)

mrjob.util.**populate_option_groups_with_options**(*assignments*, *indexed_options*)
Given a dictionary mapping `OptionGroup` and `OptionParser` objects to a list of strings representation option dests, populate the objects with options from `indexed_options` (generated by `scrape_options_and_index_by_dest()`) in alphabetical order by long option name. This function primarily exists to serve `scrape_options_into_new_groups()`.

Parameters

- **assignments** (dict of the form `{my_option_parser: ('verbose', 'help', ...), my_option_group: (...)}`) – specification of which parsers/groups should get which options
- **indexed_options** (dict generated by `util.scrape_options_and_index_by_dest()`) – options to use when populating the parsers/groups

mrjob.util.**read_file**(*path*, *fileobj=None*, *yields_lines=True*, *cleanup=None*)
Yields lines from a file, possibly decompressing it based on file extension.

Currently we handle compressed files with the extensions `.gz` and `.bz2`.

Parameters

- **path** (*string*) – file path. Need not be a path on the local filesystem (URIs are okay) as long as you specify *fileobj* too.
- **fileobj** – file object to read from. Need not be seekable. If this is omitted, we `open(path)`.
- **yields_lines** – Does iterating over *fileobj* yield lines (like file objects are supposed to)? If not, set this to `False` (useful for `boto.s3.Key`)
- **cleanup** – Optional callback to call with no arguments when EOF is reached or an exception is thrown.

mrjob.util.**read_input**(*path*, *stdin=None*)
Stream input the way Hadoop would.

- Resolve globs (`foo_*.gz`).
- Decompress `.gz` and `.bz2` files.
- If path is `'-'`, read from stdin
- If path is a directory, recursively read its contents.

You can redefine *stdin* for ease of testing. *stdin* can actually be any iterable that yields lines (e.g. a list).

mrjob.util.**safeeval**(*expr*, *globals=None*, *locals=None*)
Like eval, but with nearly everything in the environment blanked out, so that it's difficult to cause mischief.

globals and *locals* are optional dictionaries mapping names to values for those names (just like in `eval()`).

mrjob.util.**save_current_environment**(**args*, ***kwds*)
Context manager that saves os.environ and loads it back again after execution

mrjob.util.**save_cwd**(**args*, ***kwds*)
Context manager that saves the current working directory, and chdir's back to it after execution.

mrjob.util.**scrape_options_and_index_by_dest**(**parsers_and_groups*)
Scrapes `optparse` options from `OptionParser` and `OptionGroup` objects and builds a dictionary of `dest_var: [option1, option2, ...]`. This function primarily exists to serve `scrape_options_into_new_groups()`.

An example return value: `{'verbose': [<verbose_on_option>, <verbose_off_option>], 'files': [<file_append_option>]}`

Parameters **parsers_and_groups** (`OptionParser` or `OptionGroup`) – Parsers and groups to scrape option objects from

Returns dict of the form `{dest_var: [option1, option2, ...], ...}`

mrjob.util.**scrape_options_into_new_groups**(*source_groups*, *assignments*)
Puts options from the `OptionParser` and `OptionGroup` objects in *source_groups* into the keys of *assignments* according to the values of *assignments*. An example:

Parameters

- **source_groups** (list of `OptionParser` and `OptionGroup` objects) – parsers/groups to scrape options from
- **assignments** (dict with keys that are `OptionParser` and `OptionGroup` objects and values that are lists of strings) – map empty parsers/groups to lists of destination names that they should contain options for

mrjob.util.**shlex_split**(*s*)
Wrapper around shlex.split(), but convert to str if Python version < 2.7.3 when unicode support was added.

mrjob.util.**strip_microseconds**(*delta*)
Return the given `datetime.timedelta`, without microseconds.

Useful for printing `datetime.timedelta` objects.

mrjob.util.**tar_and_gzip**(*dir*, *out_path*, *filter=None*, *prefix=''*)
Tar and gzip the given *dir* to a tarball at *out_path*.

If we encounter symlinks, include the actual file, not the symlink.

Parameters

- **dir** (*str*) – dir to tar up

- **out_path** (*str*) – where to write the tarball too
- **filter** – if defined, a function that takes paths (relative to *dir* and returns `True` if we should keep them
- **prefix** (*str*) – subdirectory inside the tarball to put everything into (e.g. `'mrjob'`)

`mrjob.util.`**`unarchive`**(*archive_path*, *dest*)

Extract the contents of a tar or zip file at *archive_path* into the directory *dest*.

Parameters

- **archive_path** (*str*) – path to archive file
- **dest** (*str*) – path to directory where archive will be extracted

dest will be created if it doesn't already exist.

tar files can be gzip compressed, bzip2 compressed, or uncompressed. Files within zip files can be deflated or stored.

CHAPTER 3

What's New

For a complete list of changes, see CHANGES.txt

3.1 0.4.2

JarSteps, previously experimental, are now fully integrated into multi-step jobs, and work with both the Hadoop and EMR runners. You can now use powerful Java libraries such as Mahout in your MRJobs. For more information, see *Jar steps*.

Many options for setting up your task's environment (`--python-archive`, `setup-cmd` and `--setup-script`) have been replaced by a powerful `--setup` option. See the *Job Environment Setup Cookbook* for examples.

Similarly, many options for bootstrapping nodes on EMR (`--bootstrap-cmd`, `--bootstrap-file`, `--bootstrap-python-package` and `--bootstrap-script`) have been replaced by a single `--bootstrap` option. See the *EMR Bootstrapping Cookbook*.

This release also contains many bugfixes, including problems with boto 2.10.0+, bz2 decompression, and Python 2.5.

3.2 0.4.1

The `SORT_VALUES` option enables secondary sort, ensuring that your reducer(s) receive values in sorted order. This allows you to do things with reducers that would otherwise involve storing all the values in memory, such as:

- Receiving a grand total before any subtotals, so you can calculate percentages on the fly. See mr_next_word_stats.py for an example.
- Running a window of fixed length over an arbitrary amount of sorted values (e.g. a 24-hour window over timestamped log data).

The *max_hours_idle* option allows you to spin up EMR job flows that will terminate themselves after being idle for a certain amount of time, in a way that optimizes EMR/EC2's full-hour billing model.

For development (not production), we now recommend always using *job flow pooling*, with *max_hours_idle* enabled. Update your *mrjob.conf* like this:

```
runners:
  emr:
    max_hours_idle: 0.25
    pool_emr_job_flows: true
```

Warning: If you enable pooling *without* *max_hours_idle* (or cronning `terminate_idle_job_flows`), pooled job flows will stay active forever, costing you money!

You can now use `--no-check-input-paths` with the Hadoop runner to allow jobs to run even if `hadoop fs -ls` can't see their input files (see *check_input_paths*).

Two bits of straggling deprecated functionality were removed:

- Built-in *protocols* must be instantiated to be used (formerly they had class methods).
- Old locations for *mrjob.conf* are no longer supported.

This version also contains numerous bugfixes and natural extensions of existing functionality; many more things will now Just Work (see CHANGES.txt).

3.3 0.4.0

The default runner is now *inline* instead of *local*. This change will speed up debugging for many users. Use *local* if you need to simulate more features of Hadoop.

The EMR tools can now be accessed more easily via the *mrjob* command. Learn more *here*.

Job steps are much richer now:

- You can now use mrjob to run jar steps other than Hadoop Streaming. *More info*
- You can filter step input with UNIX commands. *More info*
- In fact, you can use arbitrary UNIX commands as your whole step (mapper/reducer/combiner). *More info*

If you Ctrl+C from the command line, your job will be terminated if you give it time. If you're running on EMR, that should prevent most accidental runaway jobs. *More info*

mrjob v0.4 requires boto 2.2.

We removed all deprecated functionality from v0.2:

- –hadoop-*-format
- –*-protocol switches
- MRJob.DEFAULT_*_PROTOCOL
- MRJob.get_default_opts()
- MRJob.protocols()
- PROTOCOL_DICT
- IF_SUCCESSFUL
- DEFAULT_CLEANUP
- S3Filesystem.get_s3_folder_keys()

We love contributions, so we wrote some *guidelines* to help you help us. See you on Github!

3.4 0.3.5

The *pool_wait_minutes* (`--pool-wait-minutes`) option lets your job delay itself in case a job flow becomes available. Reference: *Configuration quick reference*

The JOB and JOB_FLOW cleanup options tell mrjob to clean up the job and/or the job flow on failure (including Ctrl+C). See CLEANUP_CHOICES for more information.

3.5 0.3.3

You can now *include one config file from another*.

3.6 0.3.2

The EMR instance type/number options have changed to support spot instances:

- *ec2_core_instance_bid_price*
- *ec2_core_instance_type*
- *ec2_master_instance_bid_price*
- *ec2_master_instance_type*
- *ec2_slave_instance_type* (alias for *ec2_core_instance_type*)
- *ec2_task_instance_bid_price*
- *ec2_task_instance_type*

There is also a new *ami_version* option to change the AMI your job flow uses for its nodes.

For more information, see `mrjob.emr.EMRJobRunner.__init__()`.

The new `report_long_jobs` tool alerts on jobs that have run for more than X hours.

3.7 0.3

3.7.1 Features

Support for Combiners

You can now use combiners in your job. Like `mapper()` and `reducer()`, you can redefine `combiner()` in your subclass to add a single combiner step to run after your mapper but before your reducer. (`MRWordFreqCount` does this to improve performance.) `combiner_init()` and `combiner_final()` are similar to their mapper and reducer equivalents.

You can also add combiners to custom steps by adding keyword argumens to your call to `steps()`.

More info: *One-step jobs*, *Multi-step jobs*

***_init(), *_final() for mappers, reducers, combiners**

Mappers, reducers, and combiners have `*_init()` and `*_final()` methods that are run before and after the input is run through the main function (e.g. `mapper_init()` and `mapper_final()`).

More info: *One-step jobs*, *Multi-step jobs*

Custom Option Parsers

It is now possible to define your own option types and actions using a custom `OptionParser` subclass.

More info: *Custom option types*

Job Flow Pooling

EMR jobs can pull job flows out of a "pool" of similarly configured job flows. This can make it easier to use a small set of job flows across multiple automated jobs, save time and money while debugging, and generally make your life simpler.

More info: *Pooling Job Flows*

SSH Log Fetching

mrjob attempts to fetch counters and error logs for EMR jobs via SSH before trying to use S3. This method is faster, more reliable, and works with persistent job flows.

More info: *Configuring SSH credentials*

New EMR Tool: fetch_logs

If you want to fetch the counters or error logs for a job after the fact, you can use the new `fetch_logs` tool.

More info: `mrjob.tools.emr.fetch_logs`

New EMR Tool: mrboss

If you want to run a command on all nodes and inspect the output, perhaps to see what processes are running, you can use the new `mrboss` tool.

More info: `mrjob.tools.emr.mrboss`

3.7.2 Changes and Deprecations

Configuration

The search path order for `mrjob.conf` has changed. The new order is:

- The location specified by `MRJOB_CONF`
- `~/.mrjob.conf`
- `~/.mrjob` **(deprecated)**
- `mrjob.conf` in any directory in `PYTHONPATH` **(deprecated)**
- `/etc/mrjob.conf`

If your `mrjob.conf` path is deprecated, use this table to fix it:

Old Location	New Location
`~/.mrjob`	`~/.mrjob.conf`
somewhere in `PYTHONPATH`	Specify in `MRJOB_CONF`

More info: `mrjob.conf`

Defining Jobs (MRJob)

Mapper, combiner, and reducer methods no longer need to contain a yield statement if they emit no data.

The *`--hadoop-*-format`* switches are deprecated. Instead, set your job's Hadoop formats with `HADOOP_INPUT_FORMAT`/`HADOOP_OUTPUT_FORMAT` or `hadoop_input_format()`/`hadoop_output_format()`. Hadoop formats can no longer be set from `mrjob.conf`.

In addition to *`--jobconf`*, you can now set jobconf values with the `JOBCONF` attribute or the `jobconf()` method. To read jobconf values back, use `mrjob.compat.jobconf_from_env()`, which ensures that the correct name is used depending on which version of Hadoop is active.

You can now set the Hadoop partioner class with *--partitioner*, the PARTITIONER attribute, or the partitioner() method.

More info: *Hadoop configuration*

Protocols

Protocols can now be anything with a read() and write() method. Unlike previous versions of mrjob, they can be **instance methods** rather than class methods. You should use instance methods when defining your own protocols.

The *--*protocol* switches and DEFAULT_*PROTOCOL are deprecated. Instead, use the *_PROTOCOL attributes or redefine the *_protocol() methods.

Protocols now cache the decoded values of keys. Informal testing shows up to 30% speed improvements.

More info: *Protocols*

Running Jobs

All Modes

All runners are Hadoop-version aware and use the correct jobconf and combiner invocation styles. This change should decrease the number of warnings in Hadoop 0.20 environments.

All *_bin configuration options (hadoop_bin, python_bin, and ssh_bin) take lists instead of strings so you can add arguments (like ['python', '-v']). More info: *Configuration quick reference*

Cleanup options have been split into cleanup and cleanup_on_failure. There are more granular values for both of these options.

Most limitations have been lifted from passthrough options, including the former inability to use custom types and actions. More info: *Custom option types*

The job_name_prefix option is gone (was deprecated).

All URIs are passed through to Hadoop where possible. This should relax some requirements about what URIs you can use.

Steps with no mapper use **cat** instead of going through a no-op mapper.

Compressed files can be streamed with the cat() method.

EMR Mode

The default Hadoop version on EMR is now 0.20 (was 0.18).

The ec2_instance_type option only sets the instance type for slave nodes when there are multiple EC2 instance. This is because the master node can usually remain small without affecting the performance of the job.

Inline Mode

Inline mode now supports the cmdenv option.

Local Mode

Local mode now runs 2 mappers and 2 reducers in parallel by default.

There is preliminary support for simulating some jobconf variables. The current list of supported variables is:

- mapreduce.job.cache.archives
- mapreduce.job.cache.files

- mapreduce.job.cache.local.archives
- mapreduce.job.cache.local.files
- mapreduce.job.id
- mapreduce.job.local.dir
- mapreduce.map.input.file
- mapreduce.map.input.length
- mapreduce.map.input.start
- mapreduce.task.attempt.id
- mapreduce.task.id
- mapreduce.task.ismap
- mapreduce.task.output.dir
- mapreduce.task.partition

Other Stuff

boto 2.0+ is now required.

The Debian packaging has been removed from the repostory.

CHAPTER 4

Glossary

combiner A function that converts one key and a list of values that share that key (not necessarily all values for the key) to zero or more key-value pairs based on some function. See *Concepts* for details.

Hadoop Streaming A special jar that lets you run code written in any language on Hadoop. It launches a subprocess, passes it input on stdin, and receives output on stdout. Read more here.

input protocol The *protocol* that converts the input file to the key-value pairs seen by the first step. See *Protocols* for details.

internal protocol The *protocol* that converts the output of one step to the intput of the next. See *Protocols* for details.

mapper A function that convertsone key-value pair to zero or more key-value pairs based on some function. See *Concepts* for details.

output protocol The *protocol* that converts the output of the last step to the bytes written to the output file. See *Protocols* for details.

protocol An object that converts a stream of bytes to and from Python objects. See *Protocols* for details.

reducer A function that converts one key and all values that share that key to zero or more key-value pairs based on some function. See *Concepts* for details.

step One *mapper*, *combiner*, and *reducer*. Any of these may be omitted from a mrjob step as long as at least one is included.

Appendices

genindex

modindex

search

m